National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Implementing Bit Data Structures in Mizar-C

BY

Kathleen Kippen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Spring 1995

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01616-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kathleen Kippen

TITLE OF THESIS: **Implementing Bit Data Structures in Mizar-C**

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1995

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) K Kippen

Kathleen Kippen
822 - 112 B Street
Edmonton, AB
T6J 6W3

Date: Jan. 30/95

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Implementing Bit Data Structures in Mizar-C** submitted by Kathleen Kippen in partial fulfillment of the requirements for the degree of Master of Science.

Dr. H.J. Hoover (Supervisor)

Dr. W. Joerg (External)

Dr. P. Rudnicki (Examiner)

Dr. B. Joe (Chair)

Date: Jan. 30/95.

Abstract

One of the main goals of computer programming is to produce computer systems that are correct with respect to their specifications; in other words, to produce programs that do what they are supposed to. There are two main impediments to this goal: the lack of completeness in most program specifications, and the difficulty in verifying the correctness of programs, especially those that have not been fully specified. This thesis examines the use of constructive proofs to specify and produce correct code. Typically, a computer program is specified, implemented, and then proven correct. In this research a different approach is taken; the specification of the program is proven to be true, and the code is then extracted from this proof using the Mizar-C system, a natural deduction proof environment which extracts Lisp code from constructive proofs.

The main goal of this research was to define finite sequences of bit strings in Mizar-C. The work done to accomplish this goal is the content of this thesis. Some basic background to this work is given which provides a high-level overview of the notion of extracting programs from proofs, the Mizar-C system and the basic bit machine underlying its extracted programs. Appendices that go into greater detail about the system are provided. In order to provide the motivation for the specific work undertaken, some intuition into the process that was followed in defining the finite sequences is given. The work that was done to facilitate this definition is then discussed: the implementation of some new inference rules is outlined, a proof of an iterator function done in Mizar-C is presented, and the definition of the finite sequence of bit strings is given. A discussion of the feasibility of using the Mizar-C system as a programming tool is then given in conclusion.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to those who have helped me throughout my graduate studies.

I extend my appreciation to my supervisor, Dr. H. James Hoover, for all his support throughout the course of this work, and to the members of my committee for their evaluation of this thesis.

Thanks also:

to Dr. Piotr Rudnicki for all the discussions over coffee,

to Mira and Annette for keeping me sane,

to my family for their love,

and especially to my husband Kelly, whose endless support and encouragement made all of this possible.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Programming with Constructive Proofs . . . 2
- 1.3 Current Research . . . 3
- 1.4 Overview of Thesis . . . 3

2 Background 5
- 2.1 Interpreting Proofs as Programs . . . 5
- 2.2 The Mizar-C System . . . 7
- 2.3 The Bit Machine . . . 8

3 Basic Intuition 11
- 3.1 Extending the Basic *Bits* Functionality . . . 11
- 3.2 Attempting to Define New Types . . . 14

4 New Inference Rules 17
- 4.1 The Choice Rule . . . 17
- 4.2 Guarded Choice . . . 19
- 4.3 Definitions . . . 21
- 4.4 Induction . . . 22

5 Iterator Function 24
- 5.1 Motivation . . . 24
- 5.2 Defining the Iterator . . . 26
- 5.3 Proving the Iterator . . . 27
  - 5.3.1 Outline of the Proof . . . 28
- 5.4 Other Properties of the Iterator . . . 31
- 5.5 Problems Encountered During Proof . . . 31

6 Definition of Finite Sequences 34
- 6.1 Packets . . . 34
- 6.2 Finite Sequences . . . 35
- 6.3 Properties of Finite Sequences . . . 37

7 Conclusions **38**
7.1 Mizar-C as a Programming Tool 38
7.2 Using Formal Methods 39

A Sparse Realizability **41**
A.1 Conjunctions: $(\theta_1 \ \& \ \ldots \ \& \ \theta_n)$, $n \geq 1$ 41
A.2 Disjunctions: $(\theta_1$ or $\ldots$ or $\theta_n)$, $n > 1$ 42
A.3 Implications: $(\theta_1$ implies $\theta_2)$ 42
A.4 Iff: $(\theta_1$ iff $\theta_2)$ 44
A.5 Universally Quantified Formulae 44
A.6 Existentially Quantified Formulae 45
A.7 Case Analysis 46

B Mizar-C Inference Rules **47**
B.1 Constructive Inference Rules 47
B.1.1 Direct Rule 47
B.1.2 Equality Substitution 47
B.1.3 The *elim* Inference Rule 48
B.1.4 Universally Quantified Formulae 50
B.1.5 Implications 52
B.1.6 Iff 55
B.1.7 Reasoning by Cases 56
B.1.8 Existentially Quantified Formulae 58
B.1.9 Tuple Manipulation 60
B.1.10 Disjunction Manipulation 61
B.1.11 Conjunction Manipulation 63
B.1.12 Induction 64
B.1.13 The Choice Rule 64
B.1.14 Guarded Choice 64
B.1.15 Definitions 64
B.2 Non-constructive Inference Rules 65
B.2.1 Magic 65
B.2.2 Reverse Implication 65
B.2.3 Law of Excluded Middle 66
B.2.4 Negation Introduction 66
B.2.5 Negation Elimination 67
B.2.6 Contradiction Introduction 67
B.2.7 Contradiction 67
B.2.8 Equality Introduction 68
B.2.9 DeMorgan's Laws 69
B.2.10 Conversion Between Disjunctions and Implications 69

C Basic Bit String Extensions **71**

List of Figures

1 Example of Choice Rule without Content 18
2 Example of Choice Rule with Content 19
3 Example of Guarded Choice Rule with Content 21
4 Definition of One-to-One Binary Relation 22
5 Well-Founded Induction 23

6 Definition of Unary Naturals 25
7 Iterator Theorem 26
8 Iterator Definition 28
9 Initial Inductive Proof 28
10 Other Iterator Property Definitions 31
11 Iterator Theorems 31

12 Example of Equality Substitution Rule 48
13 Examples of Elimation Rule 50
14 Content-Free Universal Introduction 51
15 Universal Introduction with Content 51
16 Universal Elimination Example 52
17 Content-Free Implication Introduction 53
18 Type 1 Implication Introduction 53
19 Type 2 Implication Introduction 54
20 Implication Elimination 55
21 Iff Introduction 55
22 Iff Elimination 56
23 Introduction of Cases 57
24 Case Analysis 58
25 Existential Introduction on Formula without Content 58
26 Existential Introduction on Formula with Content 59
27 Existential Elimination when Formula has no Content 59
28 Existential Elimination when Formula has Content 60
29 Example of Tuple Rule 61
30 Disjunction Introduction 62
31 Disjunction Elimination 62
32 Examples of Conjunction Manipulation 63
33 Use of Magic Inference Rule 65

34 Examples of Reverse Implication Rule 66
35 Excluded Middle Inference Rule 66
36 Example of Negation Introduction 67
37 Example of Contra Rule 68
38 Equality Introduction 69
39 Examples of Demorgan Rule 70
40 Conversion between Implication and Disjunction 70

List of Tables

A.1 Implication Realizations 43
A.2 Realizations of Implication Elimination 43
A.3 Realizations of Iff Elimination 44
A.4 Realizations of Universal Formulae 44
A.5 Realizations of Universal Elimination 45
A.6 Realizations of Existential Formulae 45
A.7 Realizations of Existential Elimination 46

Chapter 1

Introduction

1.1 Motivation

One of the main goals of computer programming is to produce computer systems that are correct with respect to their specifications; in other words, to produce programs that do what they are supposed to. There are two main impediments to this goal: the lack of completeness in most program specifications, and the difficulty in verifying the correctness of programs, especially those that have not been fully specified. As part of the Mizar-C group at the University of Alberta, I have been investigating ways of overcoming these problems using formal methods of program specification and verification. Specifically, the Mizar-C group is examining the use of constructive proofs to specify and produce correct code. Typically, a computer program is specified, implemented, and then proven correct. Our research takes a different approach; the specification of the program is proven to be true, and the code is then extracted from this proof. We have been developing and using Mizar-C, a natural deduction proof environment which extracts Lisp programs from constructive proofs, to examine the feasibility of this method of programming [29].

The rigour of formal theories is not applicable to all types of programming. The extra programmer time, and thus cost, required to perform the formal proofs of specifications is not always feasible. However, there are computer systems where the cost of failure is so high as to justify the extra cost of using formal methods. Also, it is not necessary to formally prove the specifications of programs that can be proven by rigorous testing. Such programs have a behaviour that is defined in such a way that all possible actions can be tested by a good test suite.

There are systems where no such test suite exists; for example certain protocols, which depend upon a set of assumptions about the behaviour of the computer system, are not possible to exhaustively test in reality, and so must be simulated. In cases like these, formal methods can be used to prove that the protocol meets the specification under a set of specific assumptions. Formal methods have the added benefit of forcing the identification of the exact assumptions being made by the developer. My own experience using the Mizar-C system (as well as Hehner's method of refinement [11])

tells me that this is not a minor point. It is amazing how many assumptions a person brings into a given situation which formalization forces them, sometimes painfully, to deal with. Since some programming tasks lend themselves to formal methods, while others are better handled by good test procedures, it seems that a combination of formal methods and thorough testing would be a good approach to improving the quality of most systems.

1.2 Programming with Constructive Proofs

Programming in the Mizar-C system can be seen as analogous to programming in a high-level language. When programming in a higher-level language, the beginner programmer usually starts by implementing the specification in the most straightforward obvious way, which may or may not be the most efficient method. It is by understanding something about how the higher-level code is compiled into the lower-level object code, and how the memory is utilized that the programmer can make choices about *how* to implement in the higher-level language in order to effect the efficiency of the compiled code. In Mizar-C, often the obvious proof is not the one that will lead to the most efficient implementation. However, by understanding the relationship of the proof to the code that is generated, the prover can choose the proof structure that will result in the more efficient implementation. Although higher-level languages are supposed to make the programming task easier by abstracting some of the low-level details, the programmer who wants to produce efficient code must still take those details into account. The same thing applies to an implicit programming system like Mizar-C; if the programmer cares about the complexity of the extracted program he must pay attention to the details of the code extraction process.

For example, case analysis in a proof will correspond to a conditional test in the extracted program. Inductive proofs result in recursion; the use of lemmas may result in calls to other programs. Thus the structure of the proof determines the structure of the extracted program. Because of this, for a given specification one proof may be more desirable than another because it results in a better implementation. This is no different than how, for a specific programming task (for example sorting or searching), one algorithm may be preferable to another. Just as an experienced programmer learns how to choose the algorithm that is best suited to the desired task, the Mizar-C programmer learns how to choose the proof structure that corresponds to the best algorithm.

One problem with this implicit method of programming is that any reasoning about the complexity of the extracted program must be done outside of the Mizar-C system. Since the programs are not objects within the system it is not possible to talk about them directly. Although the extracted programs are correct with respect to their specifications, these specifications describe *what* the programs produce as output rather than *how* they produce it or what resources are consumed. Since the code generated by the computer can be quite difficult for humans to follow, reasoning about an extracted program's behaviour is even more challenging than usual. We

came across several examples of this while working in Mizar-C. Although we felt we had a general idea of the way the program would execute, many times we were surprised by the run-time behaviour. When a proof is quite complex it can be difficult to see the correspondence between the proof and the program's execution. Since a "good" program is not only correct but also efficient, this problem of complexity of extracted programs must be addressed before this implicit method of programming will be of any practical use.

1.3 Current Research

There is much research in the area of using constructive proof methods to realize correct computer programs. NUPRL [3] and COQ [6] are two systems that, like Mizar-C, perform implicit program extraction from constructive proofs. However, both systems are based upon constructive type theories, which results in what we feel is one of the major differences between them and Mizar-C. In type theories, a given type is defined by its construction method and can only be dismantled in one way. This prohibits using different methods to take apart a given data structure. The types defined in Mizar-C are not restricted to any particular method of construction, and thus for a given programming task, the most efficient method of access can be implemented (see Section 5 for further discussion). Neither NUPRL or COQ appears to have dealt with the problem of the complexity of the extracted programs. Manna and Waldinger have done work on program synthesis using deductive-tableau proofs and in [18] they mention the need to reason about the complexity of the synthesized programs, although no solution is provided. Some systems, such as PX[10], allow direct reasoning about the extracted programs. In PX, the programs exist as terms in the object language, and can thus be reasoned about directly.

There is other research which focuses on more explicit methods for proving program correctness. Most of these approaches use existing theorem provers to perform program verification rather than program extraction. For example, P. Rudnicki is using the MIZAR system to reason about programs written for a simple machine (the SCM machine) [25]. The Boyer-Moore theorem proving system, Nqthm, has been used to prove the correctness of a computing system known as the CLI Stack, which includes a microprocessor design, an assembler and a higher-level language; as well a proof of correctness for a small operating system kernel has been done [20]. HOL has been used in the process of hardware, software and protocol verification; as an example, it was used to partially verify the commercially-available VIPER microprocessor [2].

1.4 Overview of Thesis

My main project was to define finite sequences of bit strings in the Mizar-C system. Some basic background to this work is given in Chapter 2, which provides a high-level

overview of the notion of extracting programs from proofs, the Mizar-C system and the basic bit machine underlying its extracted programs. Appendix A provides greater detail into the realizability interpretation used to produce programs from proofs in Mizar-C, while Appendix B provides a full description of the inference rules of the system. Chapter 3 provides some intuition into the process that was undertaken in defining the finite sequences. This chapter provides the motivation for the work that was done in Mizar-C to facilitate the definition, and the remaining chapters outline this work. Chapter 4 describes my implementation of the new inference rules that were required to begin the definition of new types in Mizar-C. Chapter 5 outlines the proof of an iterator function that is used in the definition. The formal definition of finite sequences is then presented in Chapter 6, along with a discussion of the theory that was proven to support the definition. Finally, a general discussion of the feasibility of the Mizar-C system as a programming tool is given.

Chapter 2

Background

2.1 Interpreting Proofs as Programs

The goal of the Mizar-C project is to examine the use of constructive proofs of program specifications as a programming methodology. Given a program specification, a constructive proof of the existence of an object meeting the specification is performed, from which the program can be extracted. For example the formula

$$\forall x\ \exists y\ st\ Post[x,y]$$

can be read as a specification for a program which, for a given x, finds a y for which some property *Post[x,y]* is true. Mizar-C extracts the program from the constructive proof using a *realizability* interpretation, which relates logical connectives to computation.

In order to interpret proofs as programs, there must be a relationship between proof constructs and programming constructs. The following are some examples of general analogies that can be made between proof steps and program steps.

Proofs	Programs

$\forall$ intro:
let x
.
.
.
thus $P[x]$
$\overline{\forall x P[x]}$
	function declaration:
	`f(int x){`
	.
	.
	.
	`}`

$\forall$ elim: $\overline{\frac{\forall x P[x]}{P[6]}}$	**function application:** `f(6)`

Universal introduction can be viewed as introducing a function, and eliminating the universal variable is analogous to applying the function to the argument.

$\exists$ intro:

$$\frac{P[6+g(2)]}{\exists x\ P[x]}$$

results:

```
f(){
  .
  .
  .
   return 6+g(2);
}
```

$\exists$ elim:

$$\frac{\exists x\ P[x]}{\text{consider } t \text{ st } P[t]}$$

assignment:

```
t = f();
```

Existential introduction can be viewed as creating a macro that uses global arguments to compute a specific value, and the existential elimination can be viewed as assigning the value computed by the macro to a particular variable.

$\Rightarrow$ intro:

assume M
.
.
.
thus Q

$$\overline{M \Rightarrow Q}$$

use a module:

```
#include M
.
.
.
module Q
```

$\Rightarrow$ elim:

$$\frac{M,\ M \Rightarrow Q}{Q}$$

linking/loading:

```
ld Q.o -o Q -lM
```

One way to view implication introduction is as follows: given a set of facts M (a module), one can use them to produce Q (another module). Implication elimination can be then seen as using an existing module to create another module.

∨ intro:

$$\frac{P \Rightarrow R \quad Q \Rightarrow R}{P \vee Q \Rightarrow R}$$

case analysis:

```
switch(?)  {
  case P: R
  case Q: R
}
```

∨ elim:

$$\frac{P \vee Q,\ P \Rightarrow R,\ Q \Rightarrow R}{R}$$

case execution:

```
execute above code
```

These two rules relate to a decision procedure. The first creates the code to perform the decision, while the second would actually execute the procedure to produce the desired result.

induction:

let x
assume Ihyp:
$\forall y \,.\, y < x \Rightarrow P[y]$
.
.
.
thus $P[x]$

$$\frac{}{\forall z\ P[z]}$$

recursion:

```
int f(int x) {
        .
    ... f(x-1) ...
        .
    ... f(x-3) ...
        .
  return value of f(x)
}
```

An inductive proof results in a recursive function that can use the values from the recursive calls to compute the desired output. The structure of the proof would determine the base case and the recursive calls to be made.

2.2 The Mizar-C System

The Mizar-C system [29] provides an environment for doing natural deduction proofs, where code can be extracted from the constructive parts of the proof. Mizar-C implements a classical, limited second-order logic, and the language for the logic is related to that of the Mizar-MSE language [14]. The proof-checking environment is implemented using the Synthesizer Generator [24], which provides an interactive syntax-directed proof editing environment. Lisp is used as the language for the realizations that make up the extracted programs.

The idea of realizability was first proposed by Kleene [15] as a method for making the constructive content of arithmetical sentences explicit. The Curry-Howard isomorphism [4], or "propositions-as-types" principle, provided a means for interpreting

constructive logic and extracting its computational content as typed lambda calculus expressions. The realizability interpretation of Mizar-C, called *sparse realizability*, is based upon the Curry-Howard isomorphism, although sparse realizability destroys the isomorphism since it is no longer possible to convert programs back into their corresponding proofs.

Under a full realizability interpretation, every logical formula has an associated computation called its *realization*. However, not every realization has actual computational content. Some realizations have no inputs and are thus simply constants, or possibly expressions that require evaluation in order to produce the constant. Under sparse realizability, realizations corresponding to the Curry-Howard isomorphism are generated only for those formulae that have computational content. This *sparse realization* of formulae results in generated code that in a sense is optimized, since it eliminates the expressions, and hence the need to evaluate these expressions, that do not contribute to the overall computation.

The realizations that make up the sparse realizability interpretation are given in Appendix A. For other work exploring methods of optimizing extracted programs see [16] [1] [26].

In this thesis, the following conventions are used when presenting examples of statements and proofs done in Mizar-C:

- The type style *for x being Tx holds P[x]* will be used when writing examples of Mizar-C proof text.
- The type style `(LAMBDA () x)` will be used in the examples of realizations of Mizar-C proof text.

2.3 The Bit Machine

The realizability interpretation alone is not sufficient to be able to produce computation in Mizar-C. Without any other extensions, the programs extracted from Mizar-C using the realizations are simply functions — there are no objects to apply them to. The system has thus been extended with a basic data type of bit strings; these bit strings are the objects upon which all computation takes place. The axioms, along with an implementation of the basic primitives of this simple bit machine have been added to the Mizar-C system. The idea was to add the minimal set of primitives such that all other desired functionality could be derived from them within the Mizar-C system.

This basic data type consists of bit strings called *Bits*, which are essentially strings of 0 and 1 bits.

given Bits being [Any];
`{<<R-NONE:R-NONE>> }`
given 0, 1, nil being Bits rby bit-0, bit-1, bit-nil;
`{<<bit-0:bit-0>> <<bit-1:bit-1>> <<bit-nil:bit-nil>> }`

Two length predicates are provided, one for when two bit strings are of equal length, and one for when one bit string is shorter, or less than, in length than another bit string.

```
given bits_len_lt being [Bits, Bits ];
{<<R-NONE:R-NONE>> }
given bits_len_eq being [Bits, Bits ];
{<<R-NONE:R-NONE>> }
```

Two primitive operations, *cat* and *split*, are provided. The *cat* function concatenates two bit strings together, while the *split* function divides a bit string into two pieces modulo another bit string. For example:

$$(cat < 11, 00 >) = 1100$$
$$(split < 1101011, 1110 >) = < 1101, 011 >$$

```
given cat being (<Bits, Bits  > -> Bits) rby bit-cat;
{<<bit-cat:bit-cat>> }
BA_cat: (for x, y being Bits holds
  (ex z being Bits st (z = (cat <x, y >)))) rby bits-ba-cat;
{<<bits-ba-cat:bits-ba-cat>>}
given split being (<Bits, Bits > -> <Bits, Bits >) rby bit-split;
{<<bit-split:bit-split>> }
BA_split: (for x, y being Bits holds
  (ex z1, z2 being Bits st ((split <x, y >) = <z1, z2 >))) rby bits-ba-split;
{<<bits-ba-split:bits-ba-split>>}
BA_split_1: (for x being Bits holds ( ((split <1, x >) = <1, nil >) or
  ((split <1, x >) = <nil, 1 >) )) rby bits-ba-split-1;
{<<bits-ba-split-1:bits-ba-split-1>>}
```

A decider function is provided that determines if a given bit string is *nil* or not.

```
BA_nil_or_not: (for x being Bits holds
  ( (x = nil) or (x <> nil) ) rby bits-ba-nil-or-not;
{<<bits-ba-nil-or-not:bits-ba-nil-or-not>>}
```

A function that decides if a given bit (i.e. a bit string of length equal to 1) is a 0 or a 1 is provided.

```
BA_len_eq_1: (for x being Bits holds
  (bits_len_eq[1, x ] iff ( (x = 1) or (x = 0) ))) rby bits-ba-len-eq-1;
{<<bits-ba-len-eq-1:bits-ba-len-eq-1>>}
```

A function that, given two bit strings x and y such that $x \geq y$, determines if $x > y$ or $x = y$ is provided.

BA_len_not_lt: (for x, y being Bits holds ((not bits_len_lt[x, y]) implies
(bits_len_lt[y, x] or bits_len_eq[x, y]))) rby bits-ba-len-not-lt;
`{<<bits-ba-len-not-lt:bits-ba-len-not-lt>>}`

A well-founded induction rule, based upon the well-founded partial order *bits_len_lt* for the *Bits* data type was implemented. Several other axioms are provided (see Appendix C for complete listing of axioms). The remaining axioms do not have any computation associated with them, and so did not require that any implementation be written for them.

Combined with function abstraction, these primitives form a basis for all other computations with *Bits*. For example, one can define the bitwise logical operators, bit indexing (see Section 3.1), etc.

Chapter 3

Basic Intuition

Our goal at this point was to bootstrap from the basic *Bits* machine to one that was more useful. Simply being able to handle sequences of *Bits* would be an improvement, as it would let us pass variable length argument lists to functions. Complex computations cannot easily be understood without some structure on the data, and no one wants to have to specify and prove everything at the bit level. One of the simplest general structures is a sequence of T, where T is any available type. Once we have sequences, we can have sequences of sequences, allowing the creation of very general structures. In order for these structures to be useful, we must be able to index them. Since these sequences can be composed of non-uniform size pieces, we must deal with the problem of accessing the variable-length components.

To accomplish this, we needed to implement some data structures on top of the basic *Bits* data type. Before this could be done, several improvements had to be made to the Mizar-C system itself. At this moment, there were no inference rules that would allow new functions to be created. The system had no capabilities for defining new types and the induction rule would not work for any type other than *Bits*. These rules had to be added to the Mizar-C system before any work could be done on implementing the bit data structures. The work done to implement these new rules in the Mizar-C system is discussed in Chapter 4. Once the new rules were in place, we could begin to define new data structures in Mizar-C.

3.1 Extending the Basic *Bits* Functionality

Since the only functionality at this point was that provided by the primitives of the basic *Bits* extension, we first decided to derive some notions that provided a higher-level view of the *Bits* data type itself.

For example, the *Bits* data type has no "natural" notion of indexing into a bit string to retrieve the individual bits. One function that was built was an indexing function, based upon the length of a bit string, that would provide random access to any bit in a bit string. The following proof provided the implementation of the *index* function.

```
/* This proof creates a function (index x i) that
   returns the i-th bit of x if there is one.
   Otherwise it returns nil. To create this function
   using the choice rule, we must prove a statement
   of the form:
      for x, for i, ex y st IndexProperty[y,x,i]     */

now
  let x be Bits;
  {<<U$R252:U$R252>> }
  now
    let i be Bits;
    {<<U$R253:U$R253>> }

    /* we know that a bit string can split into two
       pieces, modulo another bit string */

    (ex x1, x2 being Bits st
      ((split <x, i >) = <x1, x2 >)) by elim[x,i](BA_split);
    {<<U$R258:(R-APPLY (R-APPLY bits-ba-split U$R252 :tag '$ELIM2256)
      U$R253 :tag '$ELIM2257)>>}
    consider x1, x2 being Bits such that
      x1_x2: ((split <x, i >) = <x1, x2 >) by direct(_PREVIOUS);
    {<<U$R259:U$R258>>}
    {<<U$R267:(R-APPLYO (R-FIRST U$R259) :tag '$EXISTC0266)>>
      <<U$R265:(R-APPLYO
        (R-APPLYO (R-SECOND U$R259) :tag '$EXISTCF262)
        :tag '$EXISTN0264)>> }
    {<<U$R263:R-NONE>>}

    /* x1 is composed of the first i bits of x, x2 the
       remaining bits. Since we are indexing from zero,
       the i-th bit of x is the first bit of x2. So we
       split it off using a bit of length one, U1.     */

    (ex y, x3 being Bits st
      ((split <x2, U1 >) = <y, x3 >)) by elim[x2,U1](BA_split);
    {<<U$R272:(R-APPLY (R-APPLY bits-ba-split U$R265 :tag '$ELIM2270)
      U.U1$0 :tag '$ELIM2271)>>}
    consider y, x3 being Bits such that
      y: ((split <x2, U1 >) = <y, x3 >) by direct(_PREVIOUS);
    {<<U$R273:U$R272>>}
    {<<U$R281:(R-APPLYO (R-FIRST U$R273) :tag '$EXISTC0280)>>
      <<U$R279:(R-APPLYO (R-APPLYO
        (R-SECOND U$R273) :tag '$EXISTCF276) :tag '$EXISTN0278)>> }
    {<<U$R277:R-NONE>>}
```

```
   /* y is thus the i-th bit of x */

   ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
     by conj(y, x1_x2);
   {<<U$R282:R-NONE>>}
   (ex x3 being Bits st
     ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
   ) by exintro(_PREVIOUS);
   {<<U$R284:(LAMBDA () U$R279) >>}
   (ex x2, x3 being Bits st
     ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
   ) by exintro(_PREVIOUS);
   {<<U$R287:(R-LIST (LAMBDA () U$R265) (LAMBDA () U$R284))>>}
   (ex x1, x2, x3 being Bits st
     ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
   ) by exintro(_PREVIOUS);
   {<<U$R291:(R-LIST (LAMBDA () U$R267) (LAMBDA () U$R287))>>}
   thus (ex y, x1, x2, x3 being Bits st
     ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
   ) by exintro(_PREVIOUS);
   {<<U$R296:(R-LIST (LAMBDA () U$R281) (LAMBDA () U$R291))>>}
 end;
 {<<U$R297:(LAMBDA (U$R253)
   (R-LIST (LAMBDA () U$R281) (LAMBDA () U$R291)))>>}

 (for i being Bits holds
   (ex y, x1, x2, x3 being Bits st
     ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <y, x3 >) )
 )) by direct(_PREVIOUS);
 {<<U$R303:U$R297>>}

 /* the function being created returns y. Since it will
    take two arguments, choice must be applied twice.    */

 thus (ex f being (Bits -> Bits) st
   (for i being Bits holds
     (ex x1, x2, x3 being Bits st
       ( ((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <(f i), x3 >) )
 ))) by choice(_PREVIOUS);
 {<<U$R310:(R-LIST (LAMBDA () (LAMBDA ($C309)
     (R-APPLY0 (R-FIRST (R-APPLY U$R303 $C309)))))
   (LAMBDA () (LAMBDA ($C309)
     (R-APPLY0 (R-SECOND (R-APPLY U$R303 $C309))))))>>}
end;
{<<U$R311:(LAMBDA (U$R252) (R-LIST
```

```
  (LAMBDA () (LAMBDA ($C309) (R-APPLY0 (R-FIRST (R-APPLY U$R303 $C309)))))
  (LAMBDA () (LAMBDA ($C309) (R-APPLY0 (R-SECOND (R-APPLY U$R303 $C309))))
)))>>}
```

(for x being Bits holds
(ex f being (Bits -> Bits) st
(for i being Bits holds
(ex x1, x2, x3 being Bits st
(((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <(f i), x3 >))
)))) by direct(_PREVIOUS);

```
{<<U$R318:U$R311>>}
```

(ex f being (Bits -> (Bits -> Bits)) st
(for x being Bits holds
(for i being Bits holds
(ex x1, x2, x3 being Bits st
(((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <((f x) i), x3 >)
))))) by choice(_PREVIOUS);

```
{<<U$R326:(R-LIST (LAMBDA () (LAMBDA ($C325)
    (R-APPLY0 (R-FIRST (R-APPLY U$R318 $C325)))))
  (LAMBDA () (LAMBDA ($C325) (R-APPLY0 (R-SECOND (R-APPLY U$R318 $C325)))
)))>>}

/* give the function the name index */
```

consider index being (Bits -> (Bits -> Bits)) such that index:
(for x being Bits holds
(for i being Bits holds
(ex x1, x2, x3 being Bits st
(((split <x, i >) = <x1, x2 >) & ((split <x2, U1 >) = <((index x) i), x3 >))
))) by direct(_PREVIOUS);

```
{<<U$R327:U$R326>>}
{<<U.index$0:(R-APPLY0 (R-FIRST U$R327) :tag '$EXISTC0334)>> }
{<<U.index$P:(R-APPLY0 (R-SECOND U$R327) :tag '$EXISTCF333)>>}
```

The function created, *(index x i)* , is a function that returns the i-th bit of x. Many other properties of the *Bits* data type were also proven.

3.2 Attempting to Define New Types

The next problem was how to define new types (i.e. structures) in Mizar-C, since we do not have the ability to give recursive structural definitions. The first new type that we defined was that of unary naturals, called *unats*, which we defined as bit strings where every bit is a 0. We can do this definition directly as:

unat[x] ≡ for y being Bits holds bits_len_lt[y, x] implies (index x y) = 0;

If we had the capability, we could have recursively defined the properties possessed by *unats* as:

unat[nil] & for x being Bits holds unat[x] implies unat[(cat <0, x >)]

Note that recursive definition of *unat[x]* is not required, since by using the universal quantifier in the direct definition, we can iterate the test for a 0-bit over each bit position (after adjusting for the length). Once we have the direct definition, we can prove the recursive properties of *unats* within the Mizar-C system.

Our next step was to try and generalize this idea to define other recursive structures. For example, to define a sequence of 2-bit chunks (instead of just 1-bit chunks), we could first derive a *2-index* function as

for x, i being Bits holds
(2-index x i) = (cat <(index x (cat < i,i >)), (index x (cat < i,(cat <i,0 >)>))>)

This function essentially computes the indices of the bits that make up the desired 2-bit chunk. The *index* function is then used with these indices to obtain the desired bits, which are then be concatenated together and returned. Using this *2-index* function, we can now define recursive structures made up of 2-bit chunks. For example, a structure called an *OddArray*, consisting of odd value 2-bit entries, could be defined as:

OddArray[x] = for i being Bits holds
(2-index x i) = (cat <0,1>) or (2-index x i) = (cat <1,1>);

In general, we could derive a function *dup* where

$$(dup\ x\ i) = \underbrace{x \cdot x \cdots x}_{(length\ i)\ times}$$

This would allow us to index into arrays of elements of arbitrary (yet fixed) size without having to create a new index function for each one. Instead, the following function would allow us to index into any structure composed of k-bit chunks.

(k-index x i k) = (first (split< (second (split <x, (dup k i)>)), (dup 0 k)>))

This function first computes a $(k * i)$-length bit string using *dup*. It then splits x with this string to create a pair of bit strings *<z1, z2>*, where *z1* is composed of the first $(k * i)$ elements of x, and *z2* is composed of the remaining elements of x. The i-th element that we want is now the first element of the bit string *z2*. Using *dup* to compute a bit string of length k, *z2* is then split with this k-length string to create another pair of bit strings, of which the first one is the desired i-th element.

The *dup* function can construct its result relatively quickly. Thus splitting off the proper $(k * i)$-size chunk from the front of the structure to get immediately at the desired location is faster than removing the k-bit chunks one at a time.

However, suppose that the chunks that make up the structure are not of a fixed size. The above solution will no longer provide a means for indexing into the structure. This is the problem that we face in trying to define finite sequences of *Bits*. In order to tell where the first chunk (or bit string) in the sequence ends, we need a prefix code,

and this is provided by the use of *Packets* to encode a bit string (see Section 6.1 for details). In the *Packet* definition, the unary naturals are used as the prefix code, to indicate the length of the bit string being encoded. This method was chosen because it was simple: it is easy to delimit a string of all 0's using a 1-bit, and this makes it simple to retrieve the prefix of the *Packet*.

A fixed length sequence of *Packets* is thus our finite sequence. In order to define the finite sequences directly, we still need some method of actually removing i packets from a finite sequence. Also, we must be able to state that each chunk indexed is indeed a bona-fide *Packet*. We must provide a method that will allow us to move iteratively through the finite sequence structure, applying the packet-test to each element, similar to how the universal quantifier was used to apply the 0-bit test to each bit position in the unary naturals. We obtain this method by proving the existence of an iterator function, which will serve as an all-purpose indexing function. The proof of the iterator function is outlined in Chapter 5. The formal definition of the finite sequences using the iterator function is discussed in Chapter 6.

Chapter 4

New Inference Rules

As previously stated, my main project in Mizar-C was to define finite sequences of bits. Before work on this could be started, several inference rules had to be added to Mizar-C. In its current state, there were no inference rules that would allow new functions to be created. All functions were disguised in the form

$$\forall x\ \exists y \ldots$$

and so the functions could not be named. The *choice* and *guarded choice* rules were implemented to provide this naming ability. The *definition* inference rule was added to allow the abbreviation of long formulae and the definition of new types (i.e. unary predicates). The current induction rule would not work for any type other than *Bits*, so a proper well-founded induction rule had to be implemented. Once these rules were in place, work could begin on defining new types in the system. The following sections outline the manner in which these inference rules were implemented.

4.1 The Choice Rule

The choice rule allows the introduction of a function from a proof of its specification. Its proper form should take a statement of existence, plus one of uniqueness:

$$\frac{\textit{for } x \textit{ being } Tx \textit{ holds } (\textit{ex } y \textit{ being } Ty \textit{ st } P[x,y]),\quad \textit{for } x \textit{ being } Tx \textit{ holds } (\textit{for } y,\ z \textit{ being } Ty \textit{ holds } P[x,y] \ \&\ P[x,z] \textit{ implies } y = z)}{\textit{ex } f \textit{ being } Tx \to Ty \textit{ st } (\textit{for } x \textit{ being } Tx \textit{ holds } P[x,\ (f\ x)])}$$

Currently realizations in Mizar-C are deterministic, so the current implementation of this rule does not require the proof of uniqueness since the same y is produced by the realization regardless of uniqueness. However, future versions of Mizar-C must add a proof of uniqueness requirement to this rule and the guarded choice rule (Section 4.2), if the underlying implementation should be changed.

There are two forms of realization for the *choice* rule. If the formula *P[x,y]* has content, then the realization is a list of two items:

1. the realization of the newly introduced function *f*, which when applied to a given *x* will return the corresponding *y*.

2. the realization of a function that, when applied to a given *x*, returns the realization of the formula *P[x, (f x)]*.

If the formula *P[x,y]* does not have content, then the realization is simply that of the new function *f*.

```
now
  let x be Any;
  {<<choice$R3:choice$R3>> }

  (x = x) by eqintro();
  {<<choice$R4:R-NONE>>}
  thus (ex y being Any st (x = y)) by exintro(_PREVIOUS);
  {<<choice$R6:(LAMBDA () choice$R3) >>}
end;
{<<choice$R7:(LAMBDA (choice$R3) (LAMBDA () choice$R3) )>>}

(for x being Any holds (ex y being Any st (x = y))) by direct(_PREVIOUS);
{<<choice$R10:choice$R7>>}

(ex f being (Any -> Any) st (for x being Any holds (x = (f x))))
  by choice(_PREVIOUS);
{<<choice$R14:(LAMBDA () (LAMBDA ($C13)
  (R-APPLY0 (R-APPLY choice$R10 $C13))))>>}
```

Figure 1: Example of Choice Rule without Content

Figure 1 illustrates the creation of an identity function. In Figure 1, since the body of the existential expression has no content, the theorem is realized by a function that will compute the *y* that is equal to *x*, for a given *x*.

Figure 2 is a contrived example in which the postcondition of the function has content. In Figure 2, the body of the expression has content, so the result is a list of the two realizations as described above.

```
now
  let x be Any;
  {<<choice$R43:choice$R43>> }

  (x = x) by eqintro();
  {<<choice$R44:R-NONE>>}
  ( (x = x) or (x ≠ x) ) by disjintro(_PREVIOUS);
  {<<choice$R45:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
  thus (ex y being Any st ( (x = y) or (x ≠ y) )) by exintro(_PREVIOUS);
  {<<choice$R47:(R-LIST (LAMBDA () choice$R43)
    (LAMBDA () choice$R45))>>}
end;
{<<choice$R48:(LAMBDA (choice$R43) (R-LIST
  (LAMBDA () choice$R43) (LAMBDA () choice$R45)))>>}

(for x being Any holds (ex y being Any st ( (x = y) or (x ≠ y) )))
  by direct(_PREVIOUS);
{<<choice$R51:choice$R48>>}

(ex f being (Any -> Any) st (for x being Any holds
  ( (x = (f x)) or (x ≠ (f x)) ) )) by choice(_PREVIOUS);
{<<choice$R55:(R-LIST
  (LAMBDA () (LAMBDA ($C54) (R-APPLY0
    (R-FIRST (R-APPLY choice$R51 $C54)))))
  (LAMBDA () (LAMBDA ($C54) (R-APPLY0
    (R-SECOND (R-APPLY choice$R51 $C54))))))>>}
```

Figure 2: Example of Choice Rule with Content

4.2 Guarded Choice

The guarded choice rule allows the introduction of a partial function.

$$\frac{\textit{for } x \textit{ being } Tx \textit{ holds } Guard[x] \textit{ implies } (\textit{ex } y \textit{ being } Ty \textit{ st } P[x,y])}{\textit{ex } f \textit{ being } Tx \rightarrow Ty \textit{ st } (\textit{for } x \textit{ being } Tx \textit{ holds } Guard[x] \textit{ implies } P[x, (f\ x)])}$$

Note that the *Guard[x]* defines the actual domain of the function *f*, yet the function produced by this guarded choice rule is said to be defined on the entire domain type *Tx*. The function *f* really has type *Guard[x]* $\rightarrow$ *Ty*, where *Guard[x]* defines a subset of the type *Tx*. If the *Guard[x]* has no content, then this rule is safe within the system, because the guard cannot in any way be required in the computation of the function *f*. Thus it is safe to *consider f* without the guard being true, since we actually have the function *f* even without the guard. In order to be able to reason about the application of *f* to *x*, we would have to perform implication elimination on the formula, which requires that *Guard[x]* be true. This makes it impossible to use the function outside

of the safe context, since we cannot introduce an application without referring to a positive occurrence of it (see Appendix B.1.3 for definition of positively occuring terms).

Trouble arises when the *Guard[x]* has content. As mentioned above, it is now possible that the guard is required as part of the computation of *f*, which means that the function *f* actually requires this guard as an argument. Without the guard, not only do we not actually *have* the function *f*, but it is not even of the type

$$Tx \to Ty$$

Instead, it has the type

$$T_{guard} \to (Tx \to Ty)$$

Since we cannot even state partial functions of this latter type in Mizar-C, *guarded choice* is only allowed on formulae where the guard has no content.

When we allow content-free definitions to be used as types, then we will be able to correctly type the function *f*, that is:

ex f being Guard → Ty st (for x being Guard holds P[x, (f x)])

However, we still must prevent definitions with content from being used, since the same problem occurs as before. In the current version of Mizar-C, only predicates defined on the type *Any* can be used as types.

The realization for expressions created through the *guarded choice* rule are the same as that for the *choice* rule.

```
now
  let x be Any;
  {<<gchoice$R112:gchoice$R112>> }
  assume P[x ];
  {<<gchoice$R92:gchoice$R92>>}
  (x = x) by eqintro();
  {<<gchoice$R72:R-NONE>>}
  ( (x = x) or (x ≠ x) ) by disjintro(_PREVIOUS);
  {<<gchoice$R73:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
  thus (ex y being Any st ( (x = y) or (x ≠ y) )) by exintro(_PREVIOUS);
  {<<gchoice$R114:(R-LIST (LAMBDA () gchoice$R112)
    (LAMBDA () gchoice$R73))>>}
end;
{<<gchoice$R115:(LAMBDA (gchoice$R112) (R-LIST
  (LAMBDA () gchoice$R112) (LAMBDA () gchoice$R73)))>>}

(for x being Any holds (P[x] implies
  (ex y being Any st ( (x = y) or (x ≠ y) )))) by direct(_PREVIOUS);
{<<gchoice$R125:gchoice$R115>>}

(ex f being (Any -> Any) st (for x being Any holds
  (P[x] implies ( (x = (f x)) or (x ≠ (f x)) )) )) by gchoice(_PREVIOUS);
{<<gchoice$R159:(R-LIST
  (LAMBDA () (LAMBDA ($C158) (R-APPLY0
    (R-FIRST (R-APPLY gchoice$R125 $C158)))))
  (LAMBDA () (LAMBDA ($C158) (R-APPLY0
    (R-SECOND (R-APPLY gchoice$R125 $C158))))))>>}
```

Figure 3: Example of Guarded Choice Rule with Content

4.3 Definitions

The *define* construct in Mizar-C permits the definition of new predicates in the following manner:

label : *define* Predicate Name *of* variable list *by* Formula ;

where variable list is of the following form:

variable names *being* variable type, ..., variable names *being* variable type

The *definition* inference rule allows definitions to be expanded or contracted, according to their specification. Definitions can be equated to macros, in that they take parameters and deliver a formula instantiated with the given parameters. For example, Figure 4 defines what it means for a binary relation to be one-to-one. If R is a

```
one-to-one: define OnetoOne of Relation being [Any, Any ] by
  (for x, y1, y2 being Any holds
       Relation[x,y1] & Relation[x,y2] implies y1 = y2) &
    (for x1, x2, y being Any holds
         Relation[x1,y] & Relation[x2,y] implies x1 = x2)
```

Figure 4: Definition of One-to-One Binary Relation

2-place predicate, then *one-to-one[R]* means that R is one to one.

The type of parameters in a definition can be dependent upon the type of other parameters, with the restriction that the dependent type be preceded in the definition by the parameter upon which it depends.

Being able to contract definitions makes the proofs easier to read, and reduces the amount of writing to be done by the user. One problem is that the definitions are in a sense opaque to the system; since the system does not automatically expand all definitions, it is not possible for it to determine whether or not the closed predicate has computational content just by looking at it. To overcome this, Mizar-C internally marks definitions as to their content.

4.4 Induction

Induction is allowed on any type using any *well-founded partial order* for that type. A *well-founded partial order* is a binary relation that produces no infinite descending chains. A binary relation is a *partial order* if it is irreflexive and transitive, which guarantees that there are no loops. However, being a partial order is not sufficient to guarantee that recursion can be performed over the ordering, since it alone does not prevent infinite chains.

In order to prevent an infinite descending chain, we not only need that there are no loops, but that for every element in the type, all sequences induced by the ordering that start at the element lead to some *minimal element*. We define *minimal element* as an element of the type for which no element is "less than" it. You can also think of "less than" as "simpler than" in terms of the construction of the elements. In the case of the recursion, these *minimal elements* can be seen as the stopping cases (or as the base cases of the induction).

In Mizar-C, the form of the induction rule is that of strong induction, as follows:

$$\frac{\begin{array}{c}\textit{WellFounded[TYPE, LT],}\\ \textit{for x being TYPE holds (for y being TYPE holds LT[y,x] implies P[y]) implies P[x]}\end{array}}{\textit{for z being TYPE holds P[z]}}$$

For this definition of induction to work, we need at least one built-in type for which induction holds. In Mizar-C, we know that the order *bits_len_lt* is well-founded over the built-in type *Bits* (Section 2.3). In order to do induction over a type using a given

ordering, a proof that the order is well-founded must be provided. The definitions and theorems stated in Figure 5 are provided in the *Bits* extension to Mizar-C to facilitate this.

WellFoundedDef: given WellFounded of S being [Any], LT being [S,S];
WellFoundedBits: WellFounded[Bits, bits_len_lt];
WellFoundedTheorem:
  for I being [Any], lt being [I,I] holds WellFounded[I,lt] implies
    (for S being [Any], f being S → I, new-lt being [S,S] holds
      (for s1, s2 being S holds (new-lt[s1,s2] iff lt[(f s1), (f s2)])) implies
WellFounded[S, new-lt]);

Figure 5: Well-Founded Induction

The *WellFoundedTheorem* states that, given a well-founded order lt_1 on some type T_1, for any other type T_2 and order lt_2 if there exists a total function that maps elements from T_2 to T_1 in such a way that the ordering of elements under lt_2 is the same as the ordering of the mapped elements under lt_1, then lt_2 is also a well-founded ordering. Since we have one ordering for which it is true, the *WellFoundedBits*, we can prove other well-founded orderings by finding a function that maps between the new type and the *Bits* type.

Chapter 5

Iterator Function

5.1 Motivation

In Mizar-C, we want to be able to define recursive structures, however if we allow recursive definitions we have to be very careful to check that the recursion used in the definition is *well-founded*. Otherwise it is possible to write definitions that will lead to contradictions. For example, from the following definition:

define P of x being Any by (not P[x]);

it is possible to prove

for x being Any holds (P[x] iff not P[x]);

In systems such as Hehner's [11], where you must explicitly implement the definitions, this is not a real problem since it is not possible to implement such non-sensical definitions. In Mizar-C however, where computation is extracted from the proofs (i.e. it is implicit), allowing such definitions can lead to extractions with dangerous behaviour. One solution to this problem is to require that, within a definition, all references to the type being defined must be made on objects that are "smaller" than the current object. It seems to be sufficient that this ordering on the objects be a *partial order* with a bottom element. Thus we could allow recursive definitions as long as a proof of the partial order was supplied. Another solution is to define recursive structures directly, and give an iterative test for correctness of construction. This can be seen as definition by implementation, which will prevent non-existent types from being defined. In order to provide an iterative test for any form of recursive structure, we need the *iterator function*.

The iterator is a function that takes a function f, composes f with itself n times, and applies the composition to a supplied argument x. Thus the three arguments to the iterator are:

- f - the function to compose.

- n - the number of times to apply the function to itself.
- x - the argument to apply the composition to.

The iterator function enables us to define recursive structures (objects) without the use of recursion in the definition; instead the structure is defined iteratively. The iterator function provides a mechanism for moving through the structure of the object, allowing us to describe what the structure looks like.

As discussed in Chapter 3, this idea of having an all-purpose indexing function occured when we were deciding how to define unary naturals in Mizar-C. The type *unat* was formally defined on the *Bits* data type as:

```
unat_def: define unat of x being Bits by
  (for y being Bits holds bits_len_lt[y, x] implies (index x y) = 0);
```

Figure 6: Definition of Unary Naturals

In other words, a *unat* is defined as a bit string of all 0's. At this point we saw that although *unats* are recursive objects, that is a *unat* is either *nil* or a *unat* concatenated with 0, it was not necessary for us to use recursion in order to define them. In the definition, the universally quantified variable acts as an index into the *unat* structure; thus the universal quantifier provides the mechanism for iterating over the structure. This works fine for *unats* where we are defining them one bit at a time. The problem arises in more complicated structures, where the elements of the structure can be sequences of arbitrary numbers of bits. We wanted a general method for using the universal quantifier to index recursive structures. Generally, recursive structures have at least two functions defined for them: one that returns the first element, and one that returns what remains after the first element is removed. Thus the iterator function provides a general method for indexing into a recursive structure using *unats* as the index.

Once we have the direct definition, we can then prove the recursive properties of the defined structure. To prove that an object meets the definition we will first have to construct the object. Often a particular recursive structure will have more than one possible method of construction, with each construction method providing a different recursive property. By defining the object directly we are not prescribing any particular construction mechanism. Instead we can prove that different methods of construction create objects of the same type, since what matters is that once constructed, these objects have a structure that conforms with the direct definition using the iterative test for construction correctness. This gives us extensional equality between the objects.

Because of this any theorems that are proven for the defined structure can be used by any of the objects that meet the definition, regardless of how they were constructed. This will allow us to dismantle objects in a different manner than they were originally

constructed. Thus the way we access the data structure can be tailored to provide the most efficient algorithm for a given task. The *Bits* data type is an example of this (Section 2.3). It is possible to dismantle *Bits* one bit at a time, by splitting a bit string *x* with the bit string *1* or *0*. It is also possible to perform a divide-and-conquer strategy on *x*, by splitting it with some arbitrary bit string *y* whose length is greater than 1. Depending upon the task, one method may provide a more efficient implementation than the other.

5.2 Defining the Iterator

The following is the theorem proving the existence of the iterator function in Mizar-C:

```
for Domain being [Any ] holds
  ex J being (Domain → Domain) → (Bits → (Domain → Domain)) st
    for f being (Domain → Domain) holds Total[Domain,f] implies
    for n being Bits holds
    for x being Domain holds ( (J f (U.length nil) x) = x &
      (J f (U.length (cat <n, 0>)) x) = (f (J f (U.length n) x)) )
```

Figure 7: Iterator Theorem

The notation *for* is a non-constructive universal quantifier, and allows for quantification over types in Mizar-C. The first argument *f* must be a function having the same domain and range, since it is applied to its own output. For this same reason the function must be total, and so the theorem is guarded by the definition *Total[Domain, f]*.

The second argument *n* is the number of times that the function *f* is to be composed with itself. In the Mizar-C system, the base type is *Bits*, which are bit strings of 0's and 1's. All other defined types must be based upon this type. For the iterator, we really need *n* to be of type *unat* (as defined in Figure 6). One problem was that, at this point in the evolution of Mizar-C, it was not possible to use types that were introduced through definitions to type term expressions. This prevented us from defining the type of the iterator as

$$(Domain \rightarrow Domain) \rightarrow (unat \rightarrow (Domain \rightarrow Domain))$$

One solution would have been to guard the entire expression with an implication whose antecedent is *unat[n]*, but instead we chose to define the iterator on the length of *n*. A function *(length n)* was proven for *Bits* that returns the *unat* that represents length of the given bit string *n*. Thus the iterator is actually defined on *unats*. Since *unats* represent the length of a bit string, the expression *(length (cat <n, 0>))* gives us the successor of the *unat* represented by *n*.

5.3 Proving the Iterator

We had several goals in trying to prove the iterator function:

1. We wanted to see what it would be like to do such a large proof in the Mizar-C system.

2. We wanted the iterator function in order to do recursive definitions without the need for recursion.

Up until this point, most of the proofs that had been done in Mizar-C were fairly small. Proving and extracting the iterator function would test most of the basic premises of the system. As hoped, attempting to do the proof pointed out some errors that existed in Mizar-C; as well it helped show us some of the things that were incomplete or needed to be added to the system. For example, we needed to add the choice and guarded choice rules in order to create the function at all. Some of the problems of using definitions as types was brought forward, as well we were forced to deal with the problem of how to implement dependent types.

5.3.1 Outline of the Proof

An iterator function was defined as having the following properties:

```
IterPropDef:
  define IterProp of Domain being [Any ],
    I being ((Domain → Domain) → (Bits → (Domain →Domain))) by
      (for f being (Domain → Domain) holds Total[Domain, f] implies
      (for n being Bits holds (for x being Domain holds
        ( ((((I f) (U.length nil)) x) = x) &
        ((((I f) (U.length (cat <n, 0 >))) x) = (f (((I f) (U.length n)) x))) )
      ))));
```

Figure 8: Iterator Definition

We started out by proving the existence of a function, which I will call the *slow iterator* which has the iterative properties. The only way to introduce a new function in Mizar-C is to use the choice or guarded choice inference rules, so using induction over n we proved:

```
for Domain being [Any ] holds
  for f being (Domain → Domain) holds Total[Domain,f] implies
  (for n being Bits holds (ex I being (Bits → (Domain → Domain)) st
    ( (for x being Domain holds
      (I (U.length nil) x) = x) &
      (for j being Bits holds
        (bits_len_lt[j, n] implies
          (for x being Domain holds
            (I (U.length (cat <j, 0>)) x) = (f (I (U.length j) x)))) &
        (not bits_len_lt[j, n] implies
          (for x being Domain holds (I (U.length (cat <j, 0>)) x) = x))
  ))))
```

Figure 9: Initial Inductive Proof

Mizar-C's induction forces a least fixed point style; that is, for each n there is an iterator function that works for all $m < n$. From this theorem we used choice to obtain:

```
(ex C being (Bits → (Bits → (Domain → Domain))) st
  (for n being Bits holds ( (for x being Domain holds
    ((((C n) (U.length nil)) x) = x) ) &
    (for j being Bits holds
```

```
( (bits_len_lt[j, n ] implies
   (for x being Domain holds
      ((((C n) (U.length (cat <j, 0 >))) x) = (f (((C n) (U.length j)) x))))) &
((not bits_len_lt[j, n ]) implies
   (for x being Domain holds
      ((((C n) (U.length (cat <j, 0 >))) x) = x)))
))))
```

The result is a proof of the existence of a function *(C n)*, but this is not quite what we want as the function itself is dependent upon the value *n*. We next proved the existence of a function *I* that is equal to the function *(C (cat <n,0>))*, for all *n*, essentially renaming the function.

```
(ex I being (Bits → (Domain → Domain)) st
  (for m being Bits holds
    (for x being Domain holds
      (((I m) x) = (((C (cat <m, 0 >)) (U.length m)) x)))))
```

The iterator properties were then proven for this new function *I*, and by performing choice one more time we get:

```
for Domain being [Any ] holds
  (ex I being ((Domain → Domain) → (Bits → (Domain → Domain))) st
    (for f being (Domain → Domain) holds
    Total[Domain,f] implies
    (for m being Bits holds (for x being Domain holds
      ( ((((I f) (U.length nil)) x) = x) &
      ((((I f) (U.length (cat <m, 0 >))) x) = (f (((I f) (U.length m)) x))) ))))))
```

This function *I* became the *slow iterator*. Because of the way that the proof of this function was done it performs some unnecessary work, mainly extra case analysis, that results in a less efficient implementation of the iterator than is possible. When executing *I*, since *I* was created by proving it to be equal to *(C (cat <n,0>))*, the first thing that is done is that *(C (cat <n,0>))* is built. This results in a very large lambda expression being created. Also the manner in which the proof of *(C (cat <n,0>))* was done causes some extra comparisons for case selection to be done when executing *(C (cat <n,0>))*. These comparisons are unnecessary from the standpoint of the run-time execution, since the actual execution path never varies when running the function. Although all the cases are needed (they are all at some point executed), the order that the cases are chosen is not affected by the inputs. Because of these extra comparisons the *slow iterator* has quadratic execution when creating the composition of *f* , and this composition is a very large lambda expression when extracted.

We know that there is an implementation of the iterator function which would build the composition of *f* in linear time. Of course, the run-time execution of applying the composition of *f* to *x* is determined by the run-time of *f* in both the *slow* and *fast* iterators. We used the fact of the existence of the *slow iterator* to

prove the existence of another function, called the *fast iterator*, which has the same properties as the *slow iterator*, but creates the composition in a linear time. Using what we know about how proofs by induction are realized in the Mizar-C system, we proved the existence of the *fast iterator* in a manner that guaranteed its linear time execution. The *fast iterator* was proven using the following induction:

```
(for n being Bits holds
  (for m being Bits holds LT[m,n] implies
    (for x being Domain holds
      (ex z being Domain st (z = (((I f) (U.length m)) x))))
  ) implies
  (for x being Domain holds
    (ex z being Domain st (z = (((I f) (U.length n)) x)))))
```

From this we can conclude the desired theorem:

```
for Domain being [Any ] holds
  (for f being (Domain → Domain) holds Total[Domain,f] implies
  (for n being Bits holds
    (for x being Domain holds
      (ex z being Domain st (z = (((I f) (U.length n)) x))))))
```

Two applications of choice, and one of guarded choice create the *fast iterator* as stated in Figure 7, which is linear in n due to the manner in which the induction was done over n (details of inductive proof given in Section 5.5).

5.4 Other Properties of the Iterator

Several other properties of the iterator function were needed when using the iterator in the definition of finite sequences. The need for the properties given in the following definitions came to light when attempting to prove properties of the finite sequences.

```
IterDistDef: define IterDist of Domain being [Any ],
  I being ((Domain -> Domain)-> (Bits -> (Domain -> Domain))) by
    (for f being (Domain -> Domain) holds Total[Domain,f] implies
      (for n being Bits holds
      (for x being Domain holds
        ((((I f) (U.length n)) (f x))=(f (((I f) (U.length n)) x)))))));

IterCompDef: define IterComp of Domain being [Any ],
  I being ((Domain -> Domain)-> (Bits -> (Domain -> Domain))) by
    (for f being (Domain -> Domain) holds Total[Domain,f] implies
      (for n being Bits holds (for m being Bits holds
      (for x being Domain holds
        ((((I f) (U.length n)) (((I f) (U.length m)) x))= (((I f) (U.length (cat <n, m>)) ) x)))))));
```

Figure 10: Other Iterator Property Definitions

It was then proven that any function for which the *IterPropDef* (as given in Figure 8) is true also has the above properties.

```
th1: (for Domain being [Any ] holds
  (for J being ((Domain -> Domain)-> (Bits -> (Domain -> Domain))) holds
    (IterProp[Domain, J] implies IterDist[Domain, J])))

th2: (for Domain being [Any ] holds
  (for J being ((Domain -> Domain)-> (Bits -> (Domain -> Domain))) holds
    (IterProp[Domain, J] implies IterComp[Domain, J])));
```

Figure 11: Iterator Theorems

5.5 Problems Encountered During Proof

It took us several attempts to be able to correctly state the theorem required for the initial proof of the *slow iterator* (as stated in Figure 9). At first we did not have the iterator function total (we missed the case for $j > n$). Next we realized that we needed to prove it for *(length n)*, which required the implementation of naturals as

unary bit strings. Since we had not yet decided how we were going to implement the type hierarchy, it was not possible to correctly type some of the terms. Our initial sketch of the proof was incorrect, since it required the instantiation (not just the existence) of a particular function outside of the scope where it was defined. All of these problems came to light when we attempted to do the proof in the Mizar-C system. On paper we thought we had a correct, complete proof. Not until we were forced to deal with every detail did we see the problems with our original ideas.

If our only goal in producing the iterator function was to use it in defining recursive structures, then it was not necessary to prove it constructively. Since the definitions themselves do not have any content, we do not require the content of the iterator in the definition. It is only when we prove that an object meets the definition that we actually "construct" the object. However, since the iterator function is proven constructively it is possible to use it when doing the actual constructions, and so it is a very useful function to have.

It turns out that the slow iterator is not much slower in actual execution than the fast iterator. The biggest difference between the functions is the resource consumption during execution, which is a result of how the induction was set up. In the slow iterator proof, the inductive step assumed the existence of an iterator function for $k < n$.

```
assume IH: (for k being Bits holds
  (LT[k, n ] implies (ex I being (Bits -> (Domain -> Domain)) st
    ( (for x being Domain holds (((I (U.length nil)) x) = x)) &
      (for j being Bits holds
        ( (bits_len_lt[j, k ] implies
          (for x being Domain holds
            (((I (U.length (cat < j, 0 >))) x) = (f ((I (U.length j)) x))))) &
        ((not bits_len_lt[j, k ]) implies
          (for x being Domain holds (((I (U.length (cat < j, 0 >))) x) = x) )) ) ) ) )) );
```

As mentioned before, this function I is dependent upon the value k and, as the inductive statement says, I behaves as an iterator function when applied to all values $j < k$. Thus for a given n, the slow iterator is actually a function which constructs a large lambda expression that is essentially an iterator for $n+1$, (the previously mentioned function *(C (cat <n,0>))*), since this function will behave as a proper iterator when applied to all values less than $n+1$ (a property stated in the above inductive step). This expression is then applied to n. In the inductive proof of the slow iterator, an implication which shows how to use the existence of an iterator function for n-1 to realize an iterator function for n is used along with the above inductive hypothesis. As a result of this, the slow iterator constructs the iterator function *(C (cat <n,0>))* in terms of an iterator for n-1, which is in turn constructed from n-2 and so on down to $n = nil$. The construction of this expression requires a considerable amount of memory at run time.

In contrast, the inductive step for the fast iterator assumes the existence of an object which is equal to the result of the slow iterator's computation.

assume indhyp: (for m being Bits holds
(LT[m, n] implies (for x being Domain holds
(ex z being Domain st (z = (((I f) (U.length m)) x))))));

In the induction on *n*, the fast iterator shows that applying the function *f* to an object that is equal to the result of applying the slow iterator to *n-1* results in an object that is equal to the result of applying the slow iterator to *n*. Thus, the fast iterator does not need to construct an iterator function for each input *n*; instead it constructs a sequence of function applications

(f (f (f ....

which is then applied to the base case of the induction (or bottom element of the recursion). This lambda expression is much smaller, and requires much less memory, than that of *(C (cat <n,0>))* constructed by the slow iterator.

Chapter 6

Definition of Finite Sequences

Once the iterator function was proven we wanted to use it to define some recursive structures. We began by defining finite sequences of bit strings, since these sequences can then be used as the basis for defining other recursive structures. The finite sequence definition imposes a structure on a bit string, which is the basic data type in Mizar-C; it provides a way of encoding and packaging a particular sequence of *Bits*. The particular *Bits* that are in the sequence have no more meaning than before. Other definitions can then be given to provide meaning for the particular bit strings occuring in the sequence, thus defining new recursive types.

6.1 Packets

Since a finite sequence of *Bits* is itself a bit string (or of type *Bits*), we need to be able to determine when a bit string is a finite sequence, and we need some way of being able to recover the individual bit strings that make up the sequence. We must be able to tell where one bit string in the sequence starts and another leaves off. To do this, each bit string in the sequence is "packaged" in a wrapper that will separate the bit strings in the sequence. A finite sequence of *Bits* is thus defined as a sequence of *Packets*, where a *Packet* is defined as a structure consisting of a bit string paired with its *length*. A '1' bit is used as a delimiter between the length and the bit string. For example:

- 000001xxxxx
  is the *Packet* of the bit string xxxxx, which has length 00000.
- 1
  is the *Packet* of the nil bit string, which has length nil.

The following is the formal definition of *Packet* used in Mizar-C:

```
packet_def: define Packet of x being Bits by
  (ex z1, z2 being Bits st
    ( ((split <x, (nozab x) >) = <(nozab x), z1 >) &
```

((split <z1, U.U1 >) = <1, z2 >) &
bits_len_eq[(nozab x), z2]));

where *U.U1* is the unary 1.

This definition looks quite complicated. It seems much easier to define *how* to create a *Packet* for a given bit string than it is to define *when* a given bit string is a *Packet*. To be a *Packet*, a bit string must consists of a sequence of 0's, followed by a 1 delimiter, followed by a sequence of bits that has the same length as the initial sequence of 0's. This is essentially what the definition *packet_def* says. The function *(nozab x)* takes a bit string x and returns the leading 0's of x (*nozab* stands for "number of zeros at beginning"). If there are no leading 0's, it returns nil. For example:

- *(nozab 00110000) = 00*
- *(nozab 111) = nil*
- *(nozab 000) = 000*

The following is the formal statement of *nozab* as proven in Mizar-C:

nozab: (for x being Bits holds (ex z1, z2 being Bits st
U.unat[(nozab x)] &
((split <x, (nozab x) >) = <(nozab x), z1 >) &
((z1 = nil) or ((split <z1, U.U1 >) = <1, z2 >))))

Thus in the packet definition, *(nozab x)* is the unary bit string of leading 0's and *z2* is the bit string being packaged. Thus *(nozab x) = (length z2)*.

6.2 Finite Sequences

The following is the formal definition of finite sequences used in Mizar-C:

finseq_def: define FinSeq of x being Bits by
(for n being Bits holds
(((first (((Iter rest) (U.length n)) x)) = nil) or
Packet[(first (((Iter rest) (U.length n)) x))]))

where *Iter* is an iterator function defined on the *Bits* domain. In other words, a finite sequence is a sequence of packets.

In order to define a finite sequence using the *Iter* function, the two functions *first* and *rest* had to be defined. We need some way to move through the sequence and retrieve the bit strings that make up the sequence. The following is the formal statement of *first* and *rest* as proven in Mizar-C:

for x being Bits holds (ex z1, z2, z3, z4 being Bits st
(((split <x, (nozab x) >) = <(nozab x), z1 >) &
((split <z1, U.U1 >) = <z2, z3 >) &
((split <z3, (nozab x) >) = <z4, (rest x) >) &
((first x) = (cat <(cat <(nozab x), z2 >), z4 >)))))

The function *(first x)* returns the first packet of the bit string x, if there is one. This function is total on the domain *Bits*, that is, it does not require that x be a finite sequence. *(first x)* returns what *will* be the first packet of x if x is a finite sequence. If x is not a correct finite sequence, then the bit string returned by *first* is not guaranteed to be a *Packet*. For example:

- *(first 0011000) = 00110*
  Even though the argument to *first* is not a correct finite sequence, the bit string returned is a valid *Packet*.

- *(first 100101) = 1*
  The argument is a valid sequence, and the returned bit string is the packet of the *nil* bit string.

- *(first 001) = 001*
  The argument is not a finite sequence, and the bit string returned is not a valid packet.

The function *rest* is very similar to the above. *(rest x)* returns the bit string that remains once *(first x)* has been removed from x. As before, *rest* is total on the *Bits* domain: if x is a valid finite sequence, then *(rest x)* will be a finite sequence, otherwise *(rest x)* is just some bit string. For example:

- *(rest 0011000) = 00*
  The bit string returned is not a valid finite sequence.

- *(rest 100101) = 00101*
  The argument is a valid sequence, and the returned bit string is the finite sequence that is left once the first packet is removed.

- *(rest 001) = nil*
  There is nothing left once *(first x)* is removed.

A finite sequence x is a bit string consisting of the concatenation of some natural number m of packets. If you apply the *rest* function n times to x, when $0 \leq n < m$, you will get a sub-sequence of x which is also a finite sequence by the definition of *rest*. Thus *first* of this sub-sequence will always be a packet by the definition of *first*. If $n \geq m$, then applying the *rest* function n times to x will remove all of the packets from x, returning *nil*. Since *(first nil) = nil* by the definition of *first*, the finite sequence definition is true for all naturals n.

6.3 Properties of Finite Sequences

Once finite sequences were defined, the theory of finite sequences needed to be proven. Several theorems are necessary to define the properties of finite sequences. One of the most important is the *finite sequence decider*, stated by the following theorem:

finseq_decide: (for x being Bits holds
(fseq.FinSeq[x] or (not fseq.FinSeq[x])))

Proven constructively, this theorem provides a program that decides whether a given bit string is a proper finite sequence, according to the finite sequence definition.

The attempt to prove this theorem outlined many sub-theorems that were useful to have. To start with, since finite sequences are defined in terms of *Packets*, it was necessary to prove a *packet decider* as follows:

packet_decide: (for x being Bits holds
(fseq.Packet[x] or (not fseq.Packet[x])))

It was necessary to define and prove the *IterDist* distributive property of the iterator function, given in Figure 10. Also, since finite sequences are defined in terms of the *rest* and *first* functions, it is useful to state the behaviour of these functions on some specific bit strings, such as the empty bit string *nil*. For example, the fact that the *rest* function returns *nil* no matter how many times it is applied to the bit string *nil* is stated with the following theorem:

fseq2.it_rest_nil: (for n being Bits holds
((((fseq.Iter fseq.rest) (U.length n)) nil) = nil))

The fact that *first* returns *nil* only if the given input bit string is *nil* is stated as:

fseq2.first_nil: (for x being Bits holds (((fseq.first x)=nil) iff (x=nil)))

Once the finite sequence decider was done, properties of finite sequences need to be stated and proven. The first theorem proven was one re-stating the definition of a finite sequence in its recursive form:

fseq3.finseq_thm: (for x being Bits holds
(fseq.FinSeq[x] iff
((x=nil) or (fseq.Packet[(fseq.first x)] & fseq.FinSeq[(fseq.rest x)]))))

Only one other theorem was proven for finite sequences, stating that the concatenation of two finite sequences is still a finite sequence:

fseq4.finseq_cat: (for x, y being Bits holds
((fseq.FinSeq[x] & fseq.FinSeq[y]) implies fseq.FinSeq[(cat <x, y>)]))

Once finite sequences are used in the definition of some other structure, the theorems about them that are useful will become evident. This happened for the *iterator*, *first* and *rest functions*. When trying to prove properties of objects that used these functions in their definitions, it was obvious which theorems would be useful.

Chapter 7

Conclusions

7.1 Mizar-C as a Programming Tool

After completing some proofs and extracting the code using the Mizar-C system, it became obvious to us that having a "correct" program is not always enough, if the definition of correctness does not take the complexity of the generated code into account. The system must provide some means of reasoning about the resource consumption of the extracted programs. It would be preferable if this could be done without having to look at the generated code, since this code can be quite difficult for humans to follow. Not only the run-time complexity of the code must be dealt with, but other measures of complexity such as resource consumption must be handled as well. The slow and fast iterator proofs can be seen as an example of this. Although the run-time execution is roughly the same for both programs, the slow iterator is a much larger program and uses more memory when executing. At the moment we see two possible ways of dealing with the complexity of the extracted code in Mizar-C. One method would be to attach some general complexity measure to the inference rules themselves. This would give us an upper-bound measure on the realizations produced through the use of these rules. Another method would be to introduce a complexity term into each specification which is then proven along with the other properties of the program. This method still requires that each inference rule have information about how it affects the complexity term.

It is the large gap between the specification language and the programming language that makes it difficult to reason about the extracted program. It might be easier to have the programming language as part of the term language, so that the code can be reasoned about directly within the system. Otherwise, as in Mizar-C, any reasoning about complexity must be done indirectly, and it remains to be seen if anything other that rough upper-bounds can be calculated.

It would be nice to extract into some language other than Lisp, preferably one that could be compiled for efficiency reasons.

There is definitely a need for a combination of classical and constructive reasoning. Many theorems are difficult to prove constructively, and if computation is not required

of it there seems to be no reason to force the user to come up with such painful proofs. Some specifications require theorems that cannot be proven constructively, such as when proving Markov's principle. These types of reasoning require notation to allow for non-content variables. The system should have a means for marking reasonings as constructive or non-constructive, and should enforce that these reasonings are used within proper contexts (see [18] and [28] for further discussion).

7.2 Using Formal Methods

After having performed many proofs on paper and then attempting to verify them using a proof checker, I can conclude that it is worthwhile to machine-check most proofs. Many paper proofs are incorrect in non-obvious ways, and having a machine check the proof provides the user with increased confidence in his solution. Also, formal environments force full specification of the problem, and this can point out errors or insufficiencies in specifications.

One of the most difficult aspects of doing formal proofs of specifications is the amount of detail that the user is forced to deal with. Any system whose goal is to provide an environment for doing formal proofs must improve this for the user. In the current version of Mizar-C, the system provides very little help in this area. The following are some ideas which would make it easier to do proofs in Mizar-C, or for that matter, any formal proof environment.

The implementation of *schemes* would help to eliminate some of the detail from proofs. For example, in the development of the finite sequences, I needed to prove several properties of iterator functions, given by the definitions in Figure 10. A scheme could be used for iterator functions where, once a function is proven to meet the iterator definition *IterPropDef* (Figure 8), all the other properties are automatically true by the scheme. Right now, elimination must be performed on the theorems in Figure 11 in order to get the other properties for a given function. Schemes are also useful in adding to the power of the system, in a sense schemes can be seen as extensions to the base set inference rules. For example, induction could be implemented through a scheme instead of an inference rule, and then several different inductive schemes corresponding to different types of induction could be implemented. Given that the user wants to use a particular scheme to prove a theorem, it should also be possible for the system to produce a "form" which outlines the steps of the proof, leaving the details for the user to fill in. For example, with an induction scheme, the base case and the inductive step can be deduced from the theorem to be proven. The system should be able to generate these steps for the user, reducing the amount of writing the user must do.

Another way to eliminate the amount of detail handled by the user would be to increase Mizar-C's automatic theorem proving power. As an example, often when proving theorems by cases, some of the cases are either trivial or not true. These cases could be proved by the system, eliminating the amount of writing to be done by the user and allowing him to concentrate on the meaningful parts of the proof.

Very large proofs contain many parts, or sub-theorems; once a sub-theorem has been proven, it is not necessary for the user to see the details of the sub-proof. The sub-theorem statement is usually all that is needed for the remainder of the proof, since it is not possible to reference any of the statements within the scope of the sub-theorem from outside of it anyway. To facilitate this, it should be possible within the system to "close" a proof once it is done, and "open" it if the details are wanted. This would allow a high-level outline of the proof to be more easily seen. In the case of Mizar-C, this may help the user to see what algorithm has been implemented by his proof, since this tends to correspond to high-level proof outline.

A different problem with detail exists when trying to use already proven theories, or proof modules. A decent mechanism for browsing through existing proof modules for theorems must be implemented. At the current instantiation of Mizar-C, the number of proof modules (or theories) was quite small, yet trying to find a desired theorem was already quite tedious. As more modules are created this problem will get worse, resulting in the re-proving of many existing theorems that cannot be located or whose existence is unknown. This problem of re-utilization of theories is encountered by most systems that provide large libraries of modules.

Appendix A

Sparse Realizability

The realizability interpretation of Mizar-C, called *sparse realizability*, is based upon the Curry-Howard isomorphism [4], although sparse realizability destroys the isomorphism since it is no longer possible to convert programs back into their corresponding proofs.

The following conventions are used:

- The type style *for x being Tx holds P[x]* will be used when writing examples of Mizar-C proof text.
- The type style `(LAMBDA () x)` will be used in the examples of realizations of Mizar-C proof text.
- Let `R[` θ `]` mean the realization of formula θ.
- In Mizar-C formulae with no content are realized by `R-NONE`. Formulae whose truth value is unknown in the current context are realized by `R-NIL`. (e.g. the non-proven formulae introduced through disjunction introduction).

A.1 Conjunctions: $(\theta_1$ & ... & $\theta_n)$, $n \geq 1$

Conjunctions are realized by:

- `R-NONE` if no θ_i has content.
- `R[`θ_i`]` if only one θ_i has content.
- `(R-LIST R[`θ_{ρ_1}`] ... R[`θ_{ρ_k}`])` when two or more of the conjuncts have content. Only the realizations of those formulae that have content are put in the list, thus $k \leq n$ and there is a one-to-one mapping from i to ρ_j which maps each θ_i that has content to each θ_{ρ_j} whose realizations are elements of the list.

Conjunction elimination allows the projection of one of the conjuncts from a conjunctive formula. Conjunction elimination of a conjunct θ from a conjunction ϕ is realized by:

- R-NONE if θ has no content.
- R[ϕ] if θ is the only conjunct of ϕ with content.
- (R-SELECT n R[ϕ]) if more than one conjunct of ϕ has content. The R-SELECT operator projects the *nth* element out of an R-LIST, so θ must be the *nth* conjunct of ϕ with content.

A.2 Disjunctions: (θ_1 or ... or θ_n), $n > 1$

Disjunctions are realized by:

(R-LIST (LAMBDA () R[θ_1]) ... (LAMBDA () R[θ_n]))

where R[θ_i] is actually R-NIL if the sub-formula is not known to be true in the current context. Otherwise the subexpressions exist, and their realizations are wrapped in lambda forms to delay evaluation until it is required.

In order to perform disjunction elimination, a formula must be provided that is the negation of one of the disjuncts. This disjunct is then removed from the disjunctive formula. Disjunction elimination of a disjunct θ_m from a disjunction ϕ is performed by projecting all of the disjuncts other than θ_m out of ϕ. If ϕ contains more than two disjuncts, the projections become the elements of a new disjunction ϕ'. Thus the two cases are:

- if ϕ contains only two disjuncts then the disjunction elimination of θ_m is realized by:
  - R-NONE if the remaining disjunct has no content.
  - (R-APPLY0 (R-SELECT n R[ϕ])) where the remaining disjunct is the *nth* element of the disjunction. The disjunct is unwrapped at this point, since it is no longer a disjunction, but simply a true statement.
- if ϕ contains $k > 2$ disjuncts, then the disjunction elimination of θ_m is realized by (R-LIST (R-SELECT n R[ϕ]) ...(R-SELECT n R[ϕ])) for all n where $(0 \leq n < k)$ and $(n \neq m)$.

A.3 Implications: (θ_1 implies θ_2)

Implications can be seen as providing a method that uses the facts of the antecedent to produce the consequent. From this viewpoint an implication is a function that takes the antecedent as an argument and uses it to compute the consequent. However, if the antecedent has no content, it has no computational "facts" to be used in constructing the consequent. In this case, the implication no longer needs to take the antecedent as an argument in order to compute the consequent, thus its realization can be optimized

Case #	θ_1	θ_2	Type	Realization
1	no content	no content	0	R-NONE
2	content	no content	0	R-NONE
3	no content	content	1	R[θ_2]
4	content	content	2	(LAMBDA (x) R[θ_2][R[θ_1] := x])

Table A.1: Implication Realizations

to eliminate an unnecessary function application. Implications are realized according to Table A.1.

In cases 1 and 2, since the consequent has no computational content, the implication has none either. Although the antecedent has content in case 2, since the consequent has no content it is not possible that the realization of the antecedent is required for any computation (if the consequent had made use of the antecedent then the consequent would have content). Thus it is not necessary to retain the realization of the antecedent at all.

In case 3, since the antecedent has no content it obviously cannot be involved in any of the computation that is done in the consequent, so only the realization of the consequent is necessary to realize the implication. In this case, it is most likely that the antecedent is a guard that provides a context for executing the consequent. Within the system, it will only be possible to use the consequent in contexts where the antecedent has been shown to be true, so the above realization is safe.

In case 4, where both the antecedent and the consequent have content, it is possible that the consequent uses the antecedent's realization in its own computation. Since the formula is an implication, implication elimination must be done in order to be able to use the consequent. This elimination requires that the antecedent formula be supplied. The realization of the supplied antecedent formula is bound to the variable x when the implication elimination is performed.

Implication elimination on formula θ_1 implies θ_2, is realized according to Table A.2.

θ_1 implies θ_2	Realization
Type 0	R-NONE
Type 1	R[θ_1 implies θ_2]
Type 2	(R-APPLY R[θ_1 implies θ_2] R[θ_1])

Table A.2: Realizations of Implication Elimination

The type information is used by the system when performing eliminations. Implications of type 0 have no content, and cannot be used as arguments to formulae requiring content. Implications of type 1 do not require an antecedent with content, while type 2 implications require a constructive proof of their antecedent, and need to be applied to this realization.

A.4 Iff: (θ_1 iff θ_2)

Iff formulae are realized by a conjunction of the realizations of the two implications (θ_1 implies θ_2) and (θ_2 implies θ_1), according to the rules for conjunctions.

Iff eliminations are realized according to Table A.3.

θ_1 iff θ_2	θ_1 implies θ_2	θ_2 implies θ_1	Arg	Realization
no content	type 0	type 0	$\theta_1 or \theta_2$	R-NONE
content	type 0	type 1 or 2	θ_1	R-NONE
content	type 1	type 0	θ_1	R[θ_1 iff θ_2]
content	type 1	type 1 or 2	θ_1	(R-FIRST R[θ_1 iff θ_2])
content	type 2	type 0	θ_1	(R-APPLY R[θ_1 iff θ_2] R[θ_1])
content	type 2	type 1 or 2	θ_1	(R-APPLY (R-FIRST R[θ_1 iff θ_2]) R[θ_1])
content	type 1 or 2	type 0	θ_2	R-NONE
content	type 0	type 1	θ_2	R[θ_1 iff θ_2]
content	type 1 or 2	type 1	θ_2	(R-SECOND R[θ_1 iff θ_2])
content	type 0	type 2	θ_2	(R-APPLY R[θ_1 iff θ_2] R[θ_2])
content	type 1 or 2	type 2	θ_2	(R-APPLY (R-SECOND R[θ_1 iff θ_2]) R[θ_2])

Table A.3: Realizations of Iff Elimination

A.5 Universally Quantified Formulae

for x being Type holds θ

The intuition behind the realization of universally quantified formulae is the same as that for implications. If the formula θ that is universally quantified has content, then it may require an argument x of the correct type in order to perform its computation. Thus it can be viewed as a function that takes one argument. However, if θ has no content, then it can not require the argument x as input, and its realization can be optimized to eliminate the unnecessary function application.

Universally quantified formulae are realized according to Table A.4.

θ	Realization
no content	R-NONE
content	(LAMBDA (x) R[θ])

Table A.4: Realizations of Universal Formulae

Universal elimination on a formula α, using a term ϕ, is realized according to Table A.5.

α	Realization
no content	R-NONE
content	(R-APPLY R[α] R[ϕ])

Table A.5: Realizations of Universal Elimination

A.6 Existentially Quantified Formulae

ex x being Tx st θ

Existentially quantified formulae are realized according to Table A.6.

θ	Realization
no content	(LAMBDA () R[x])
content	(R-LIST (LAMBDA () R[x]) (LAMBDA () R[θ]))

Table A.6: Realizations of Existential Formulae

There are two possible interpretations for the existential statement:

- It can be viewed as capturing an explicit value x for export.
- It can be seen as capturing a "procedure" for computing the value x. This function is then evaluated when the explicit value is requested (by using the *consider* statement).

Mizar-C has adopted the second lazy interpretation, and by wrapping the realization of x in a lambda form we have delayed the computation of the value of x until it is explicitly requested (by unwrapping it).

In the case where θ has no content, it is not important to retain its realization. The realization of the existential formula is simply the function that computes the value of the existential variable. If θ has content, the realization of the existential statement becomes a list of two elements: the first element is the function that will compute the value of the existential variable, and the second element is the computation of the formula θ.

When an existential elimination is performed on a formula ϕ, two realizations are produced: one that realizes the value of the variable, and one that realizes θ, the body of the existential formula ϕ.

In future versions of Mizar-C, it will be possible to have no-content existential formulae where the body θ of the existential may have content, but the existential variable x does not. This is necessary in order to prove, or even use, certain theorems such as Markov's principle, where non-constructive existence is used.

θ	Realization
no content	(R-APPLY0 R[ϕ]) R-NONE
content	(R-APPLY0 (R-FIRST R[ϕ])) R-APPLY0 (R-SECOND R[ϕ]))

Table A.7: Realizations of Existential Elimination

A.7 Case Analysis

Given a disjunction θ, and some implications ϕ_1 through ϕ_n, case analysis is realized by:

```
(R-CASE R[θ] (R-LIST (LAMBDA () (R-LIST Type R[φ1]))
                     ⋮
                     (LAMBDA () (R-LIST Type R[φm]))))
```

The antecedents of the implications must correspond to the disjuncts of θ, and the consequents must all match the goal formula. The *Type* information indicates the type of the implication (as described in Table A.1), which determines how the implication elimination is performed (see Table A.2). The R-CASE function evaluates the disjuncts of θ to determine which one is true, and then selects the corresponding implication to compute the goal formula. Since it is possible for more than one disjunct to be true, the present implementation of Mizar-C uses the first disjunct found to be true. However, this leaves open the possibility of allowing parallel evaluation and execution of all the true cases.

Appendix B

Mizar-C Inference Rules

Mizar-C implements a multi-sorted limited second-order natural deduction logic. It provides the following base set of inference rules:

Case Analysis	Existential Introduction	Negation Elimination
Case Introduction	Existential Elimination	Negation Introduction
Choice Rule	Guarded Choice	Quantifier Negation
Conjunction Manipulation	Iff Introduction	Reverse Implication
Contradiction	Iff Elimination	Take
Contradiction Introduction	Iff Swap	Tuple
Definitions	Implication Introduction	Universal Introduction
DeMorgan's Laws	Implication Elimination	Universal Elimination
Disjunction Introduction	Induction Rule	
Disjunction Elimination	Excluded Middle	
Equality Introduction	Magic	
Equality Substitution		

B.1 Constructive Inference Rules

B.1.1 Direct Rule

Synopsis: *direct(Formula)*
This inference rule allows previously proven results to be re-stated. It checks to see that the reference *Formula* is equal to the goal formula. It is useful for making the results of a *now* reasoning explicit. For examples of use, see the sections on Universal Introduction, Implication Introduction or Existential Elimination.

B.1.2 Equality Substitution

Synopsis: *equality*($Formula, Equality_0, \ldots, Equality_m$)
The Equality Substitution Rule performs term substitution in the *Formula* through

the use of term equalities. Any number of simple substitutions may be performed, as long as there is an equality formula *Equality*$_i$ for each substitution.

```
environ
  given x, y, z being Any;
  {<<x:R-NONE>> <<y:R-NONE>> <<z:R-NONE>> }
  A: (x = y);
  {<<A$P11:R-NONE>>}
  B: (y = z);
  {<<B$P13:R-NONE>>}

begin
  ex1: (x = z) by equality(A, B);
  {<<eq.ex1$P:A$P11>>}

  now
    (x = x) by eqintro();
    {<<eq$R0:R-NONE>>}
    thus (ex q being Any st (q = x)) by exintro(_PREVIOUS);
    {<<eq$R5:(LAMBDA () x) >>}
  end; {<<eq$R8:(LAMBDA () x) >>}
  ex2: (ex q being Any st (q = z)) by equality(_PREVIOUS, A, B);
  {<<eq.ex2$P:eq$R8>>}
```

Figure 12: Example of Equality Substitution Rule

In the example in Figure 12, in *ex1* only one substitution was performed. In *ex2*, two substitutions were done: first y for x and then z for y. The substitutions are performed in the order of the given equalities. The realization of the goal is that of the original *Formula* being substituted.

B.1.3 The *elim* Inference Rule

Synopsis: *elim[universal substitution list] (Formula*$_0, \ldots,$ *Formula*$_n$*)*, $n \geq 0$;
In Mizar-C, *elim* is a very powerful rule which performs universal elimination, implication elimination and iff elimination on the argument *Formula*$_0$. The *universal substitution list* is used for universal quantifier elimination; it lists the variables or terms that are to be substituted for the universal quantifiers of *Formula*$_0$, in the order of elimination. Any terms that are being substituted must already be defined, and their justifications (a formula where the term occurs positively) must be provided in the formula list.
A term t occurs positively in a formula ϕ when:

- ϕ is a predicate and t occurs inside of it (e.g. $\phi[\ldots, t, \ldots]$).

- ϕ is a conjunction, $\theta_0 \wedge \ldots \wedge \theta_n$, and t occurs positively in some θ_i.

Formula$_1$ through *Formula*$_n$ are either the antecedents of the implications or iffs in *Formula*$_0$, or the justifications of terms in the substitution list. These formulae must appear in the correct order of elimination. If an antecedent is a disjunction, then it is sufficient for *Formula*$_i$ to refer to one of the disjuncts. The goal formula resulting from the call to *elim* has the number of quantifications indicated by the *universal substitution list* and antecedents given by *Formula*$_1$ through *Formula*$_n$ stripped off of *Formula*$_0$. If after all eliminations, the result would be a conjunction, the goal formula can simply be one of the conjuncts. Figure 13 illustrates the many uses of *elim*. These examples do not show the realizations generated for each inference. In the sections on universal elimination, implication elimination and iff elimination that follow, the realizations that result from the use of *elim* for each case are discussed.

```
now
   assume IH: (for x being Nat holds (P[x] implies
      (for y being Nat holds ( (x = y) or (x ≠ y) );
   guard: P[px] by magic;
   { we can do simple universal elimination}
   1: P[px] implies (for y being Nat holds
      ( (px = y) or (px ≠y) ) by elim[px](IH);
   { we can do simple implication elimination}
   2: (for y being Nat holds
      ( (px = y) or (px ≠ y) ) by elim[](1,guard);
   {another universal elimination }
   3: (px = py) or (px ≠ py) by elim[py](2);
   { or we can do the eliminations (universal and implication)
   all in one step}
   4: (px = py) or (px ≠ py) by elim[px, py](IH, guard);
   {we can also eliminate using terms, by giving the
   formula where they occur positively as a justification}
   a1: P[(f a) ];
   ((f a) = (f a)) or ((f a) ≠ (f a)) by elim[(f a), (f a)](IH, a1, a1, a1);
{ the first reference to a1 is the justification of the first
   substitution term,
   the second reference to a1 is used as the antecedent of the
   implication, the third reference to a1 is the justification of
   the second substitution term.  }
end;
A1: for x being Nat holds (P[x] or Q[x]) implies R[x];
A2: P[px];
{ we can eliminate using one of the disjuncts}
R[px] by elim[px](A1, A2);
A3: for x being Nat holds P[x] iff (R[x] & T[x]);
{ we can refer to just one of the goal conjuncts}
T[px] by elim[px](A3, A2);
```

Figure 13: Examples of Elimation Rule

B.1.4 Universally Quantified Formulae

Universal introduction occurs when variables have been introduced by the *let* construction within a *now* reasoning. The conclusion of the *now* reasoning, created using the *thus* statement, is the formula that is universally quantified. The universal sentence that is created is interesting only if the *thus* conclusion references the *let*-ed variables. Note that in the *now* reasoning where this rule is invoked, there may be several variables introduced with *let* ; each variable results in a new quantification

level.

```
now
  let x,y be Any;
  {<<univ$R15:univ$R15>> <<univ$R16:univ$R16>> }
  thus x = x by eqintro();
  {<<univ$R17:R-NONE>>}
end;
{<<univ$R18:R-NONE>>}
Result1: for x, y being Any holds x = x by direct(_PREVIOUS);
{<<univ.Result1$P: R-NONE>>}
```

Figure 14: Content-Free Universal Introduction

Figure 14 shows a simple *now* reasoning where two variables are introduced with a *let* statement. Although the *thus* conclusion does not reference the variable *y*, the universal sentence created (the statement labeled *Result1*) has two nested quantifications over both variables *x* and *y*. Since the formula being quantified has no content, the universal sentence is realized by `R-NONE`.

```
now
  let x, y be Any;
  {<<univ$R19:univ$R19>> <<univ$R20:univ$R20>> }
  x = x by eqintro();
  {<<univ$R21:R-NONE>>}
  thus x = x or x ≠ x by disjintro(_PREVIOUS);
  {<<univ$R24:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
end;
{<<univ$R27:(LAMBDA (univ$R19) (LAMBDA (univ$R20)
  (R-LIST (LAMBDA () R-NONE) R-NIL)))>>}
Result2: for x, y being Any holds x = x or x ≠ x by direct(_PREVIOUS);
{<<univ.Result2$P:(LAMBDA ($ALL6) (LAMBDA ($ALL7)
  (R-LIST (LAMBDA () R-NONE) R-NIL)))>>}
```

Figure 15: Universal Introduction with Content

Figure 15 shows the realization that is created when the formula being universally quantified has content. In this case, universal quantification corresponds to function abstraction. Since there are two *let*-ed variables a function is created that takes two arguments. Conversely, function application corresponds to universal elimination, which provides the arguments for the function.

Universal elimination is done using the *elim* rule, as shown in Section B.1.3.

A: (x = x) or (x ≠ x) by elim[x,x](Result2);
`{univ$R11:(R-APPLY (R-APPLY univ.Result2$P x) x)>>}`
1: (f x) = (f x) by magic;
`{<<univ.1$P:R-NONE>>}`
B: (f x) = (f x) or (f x)≠(f x) by elim[(f x), x](Result2, 1);
`{<<univ.R31:(R-APPLY (R-APPLY univ.Result2$P (FUNCALL f x)) x)>>}`
C: (f x) = (f x) by elim[(f x), (f x)](Result1, 1, 1);
`{<<univ$R8:R-NONE>>}`

Figure 16: Universal Elimination Example

Since statements *A* and *B* in Figure 16 perform universal elimination on a formula with content, this results in a function application. The realization of the universal formula *Result2* is applied to the arguments provided. In statement *B*, a term is being substituted for one of the universal quantifiers which requires that a *justification* be given; this is provided by the reference to statement *1*. In all three cases, although the second quantified variable *y* is not present in the formula body, an argument must be provided for it anyway. In statement *C*, the term *(f x)* is being used as the argument for both quantifiers. A justification for each substitution is required, which results in two references to statement *1*.

B.1.5 Implications

Implication introduction is similar to universal introduction in the sense that it corresponds to the results of certain hypothetical reasonings. An implication is created as a result after a formula has been introduced with an *assume* statement in a *now* reasoning. The antecedent of the resulting implication corresponds to the *assumed* formula, and the consequent to the *thus* conclusion reached by the sentences after the *assume* statement.

```
Ex1: now
   assume 1: P[x ];
   {<<imp$R34:imp$R34>>}
   thus (x = x) by eqintro();
   {<<imp$R41:R-NONE>>}
end;
{<<imp$R44:R-NONE>>}

Ex2: now
   assume 1: (ex y being Any st (x = y));
   {<<imp$R53:imp$R53>>}
   thus P[x ] by direct(A);
   {<<imp$R61:A$P9>>}
end;
{<<imp$R64:R-NONE>>}
```

Figure 17: Content-Free Implication Introduction

In both *Ex1* and *Ex2* given in Figure 17, since the consequent of the implication is a formula without content, the entire implication has no content and is realized by R-NONE.

```
now
   assume 1: P[x ];
   {<<imp$R45:imp$R45>>}
   thus (ex y being Any st P[y ]) by exintro(_PREVIOUS);
   {<<imp$R49:(LAMBDA () x)>>}
end;
{<<imp$R52:(LAMBDA () x)>>}
```

Figure 18: Type 1 Implication Introduction

In Figure 18, the antecedent formula *P[x]* has no content. However, since the consequent is an existentially quantified formula it does have content, and so the realization of the implication is simply the realization of the consequent.

```
now
   assume 1: (ex y being Any st (x = y));
   {<<imp$R2:imp$R2>>}
   consider y1 being Any such that y1: (x = y1) by direct(_PREVIOUS);
   {<<imp$R11:imp$R2>>}
   {<<imp$R15:(R-APPLY0 imp$R11)>>}
   {<<imp$R13:R-NONE>>}
   P[y1 ] by equality(A, y1);
   {<<imp$R26:A$P9>>}
   thus (ex z being Any st P[z ]) by exintro(_PREVIOUS);
   {<<imp$R30:(LAMBDA () imp$R15) >>}
end;
{<<imp$R33:(LAMBDA (imp$R2) (LAMBDA () imp$R15) )>>}
result4: ((ex y being Any st (x = y)) implies
   (ex z being Any st P[z ])) by direct(_PREVIOUS);
{<<imp.result4$P:imp$R33>>}
```

Figure 19: Type 2 Implication Introduction

With all the labels expanded, the realization of the implication proven in Figure 19 is:

```
(LAMBDA (imp$R2) (LAMBDA () (R-APPLY0 imp$R2)))
```

This is an example of an implication where the realization of the antecedent formula is used in the realization of the consequent. The antecedent realization will be a function to compute the value of the existential variable *y*. Within the reasoning, this value is named *y1*. The realization of the consequent is also a function to compute an existential variable *z*, and since the value of *z* is that of *y1*, the function provided by the antecedent's realization is used to compute it. Thus the implication is realized by a lambda form that takes the realization of the antecedent formula as an argument.

Implication elimination is performed using the *elim* inference rule, as described in Section B.1.3. In order to perform implication elimination the antecedent formula must be provided. If the implication requires an antecedent formula with content, as in Figure 19, the result of the implication elimination is a realization where the realization of the implication is applied to the realization of the supplied antecedent formula.

In Figure 20, the realization of the implication `imp.result4$P` is applied to the realization of the antecedent `imp.B$P`. If the implication being eliminated does not require the realization of the antecedent, as in Figure 17 and Figure 18, then the implication is realized by the realization of the consequent; hence implication elimination does not change the realization at all.

```
(x = x) by eqintro();
{<<imp$R27:R-NONE>>}
B: (ex q being Any st x = q)) by exintro(_PREVIOUS);
{<<imp.B$P:(LAMBDA (x) x) >>}
(ex z being Any st P[z ]) by elim(result4, B);
{<<imp$R31:(R-APPLY imp.result4$P imp.B$P )>>}
```

Figure 20: Implication Elimination

B.1.6 Iff

The inference rule *iffintro* is used to introduce an iff from two implications.
Synopsis: *iffintro(Implication$_0$, Implication$_1$)*

$$\frac{\theta \rightarrow \phi, \phi \rightarrow \theta}{\theta \leftrightarrow \phi | \phi \leftrightarrow \theta}$$

```
now
   assume ( (x = x) or (x ≠ x) );
   {<<iffs$R4:iffs$R4>>}
   thus ( (x = x) or (x ≠ x) ) by direct(_PREVIOUS);
   {<<iffs$R5:iffs$R4>>}
end;
{<<iffs$R6:(LAMBDA (iffs$R4) iffs$R4)>>}

A: (( (x = x) or (x ≠ x) ) implies ( (x = x) or (x ≠ x) )) by direct(_PREVIOUS);
{<<iffs.A$P:iffs$R6>>}
B:(( (x = x) or (x ≠ x) ) iff ( (x = x) or (x ≠ x) )) by iffintro(A, A);
{<<iffs.B$P:(R-LIST iffs.A$P iffs.A$P)>>}
```

Figure 21: Iff Introduction

An iff formula is realized by a conjunction of the realizations of the implications used to form it. Thus the iff realization depends upon whether the implications have content, according to the rules given for conjunction realizations in Section A.1.

Iff elimination is performed using the *elim* inference rule. as described in Section B.1.3.

$$\frac{\theta \leftrightarrow \phi, \phi}{\theta} \quad \frac{\theta \leftrightarrow \phi, \theta}{\phi}$$

In order to perform iff elimination, either the left or right side formula of the iff must be provided. When the goal formula has no content it is realized by `R-NONE`; when the goal formula has content its realization is determined by the implications that were used when forming the iff. If the left formula θ is being eliminated, then the first implication, ($Implication_0$ in the introduction of the iff), is used and implication elimination is performed on it using θ as the antecedent. If the right formula ϕ is being eliminated, then the second implication is used ($Implication_1$ in the introduction of the iff). The realization that results follows the rules for implication elimination as outlined in Table A.2. Using iff formula B from Figure 21:

1: (x = x) by eqintro();
`{<<iffs.1$P:R-NONE>>}`
2: ((x = x) or (x ≠ x)) by disjintro(_PREVIOUS);
`{<<iffs.2$P:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}`
((x = x) or (x ≠ x)) by elim(B, 2);
`{<<iffs$R28:(R-APPLY (R-FIRST iffs.B$P) iffs.2$P)>>}`

Figure 22: Iff Elimination

B.1.7 Reasoning by Cases

Case introduction, performed with the *caseintro* inference rule, allows the antecedent of an implication to be weakened by making it a disjunction.
Synopsis: *caseintro($Implication_0$, $Implication_1$, ..., $Implication_m$)*

$$\frac{\phi_0 \rightarrow \theta, \phi_1 \rightarrow \theta, \ldots \phi_m \rightarrow \theta}{(\phi_{p(0)} \vee \phi_{p(1)} \vee \ldots \phi_{p(m)}) \rightarrow \theta}$$

for any permutation $p : \{0 \ldots m\} \rightarrow \{0 \ldots m\}$.

The resulting realization is a function, which takes a disjunction for an argument, and uses case analysis of the disjunction to determine which of the implications should be used for the computation.

```
now
  assume (x = x);
  {<<case$R0:case$R0>>}
  thus ( (x = x) or (x ≠ x) ) by disjintro(_PREVIOUS);
  {<<case$R1:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
end;
{<<case$R2:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
A: ((x = x) implies ( (x = x) or (x ≠ x) )) by direct(_PREVIOUS);
{<<case.A$P:case$R2>>}

now
  assume (x ≠ x);
  {<<case$R3:case$R3>>}
  thus ( (x = x) or (x ≠ x) ) by disjintro(_PREVIOUS);
  {<<case$R4:(R-LIST R-NIL (LAMBDA () R-NONE))>>}
end;
{<<case$R5:(R-LIST R-NIL (LAMBDA () R-NONE))>>}
B: ((x ≠ x) implies ( (x = x) or (x ≠ x) )) by direct(_PREVIOUS);
{<<case.B$P:case$R5>>}

1: (( (x = x) or (x ≠ x) ) implies ( (x = x) or (x ≠ x) )) by caseintro(A, B);
{<<case.1$P:(LAMBDA ($F6) (R-CASE $F6
  (R-LIST (LAMBDA () (R-LIST 1 case.A$P ))
          (LAMBDA () (R-LIST 1 case.B$P )))) )>>}

2: (( (x ≠ x) or (x = x) ) implies ( (x = x) or (x ≠ x) )) by caseintro(B, A);
{<<case.2$P:(LAMBDA ($F19) (R-CASE $F19
  (R-LIST (LAMBDA () (R-LIST 1 case.B$P ))
          (LAMBDA () (R-LIST 1 case.A$P )))) )>>}
```

Figure 23: Introduction of Cases

Case analysis is performed using the *caseanal* inference rule.
Synopsis: *caseanal(Disjunction, Implication*$_0$ *, ..., Implication*$_m$*)* $m \geq 1$

$$\frac{\phi_0 \vee \phi_1 \vee \ldots \phi_m, \phi_0 \rightarrow \theta, \ldots, \phi_m \rightarrow \theta}{\theta}$$

The antecedents of the *Implications* must be in 1-1 correspondence with the disjuncts in *Disjunction*. The goal formula must be the consequent of all the *Implications*. Using the implications *A* and *B* from Figure 23:

```
(x = x) by eqintro();
{<<case$R82:R-NONE>>}
3: ( (x = x) or (x ≠ x) ) by disjintro(_PREVIOUS);
{<<case.3$P:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
( (x = x) or (x ≠ x) ) by caseanal(3, A, B);
{<<case$R89:(R-CASE case.3$P
  (R-LIST (LAMBDA () (R-LIST 1 case.A$P ))
          (LAMBDA () (R-LIST 1 case.B$P ))))>>}
```

Figure 24: Case Analysis

B.1.8 Existentially Quantified Formulae

The inference rule *exintro* is used to introduce an existentially quantified variable.
Synopsis: *exintro(Formula)*
In order to introduce an existentially quantified variable with content, the existence of an actual object must be provided. The goal formula must be the *Formula*, existentially quantified, with the actual object in *Formula* replaced with the existential variable. The realization of an existentially quantified formula depends upon whether or not the formula being quantified has content.

```
x = x by equintro();
{<<exis$R0:R-NONE>>}
A: ex y being Any st (x = y) by exintro(_PREVIOUS);
{<<exis.A$P:(LAMBDA () x)>>}
```

Figure 25: Existential Introduction on Formula without Content

The realization of the existential formula created in Figure 25 is simply the function to compute the value of the existential variable y. When evaluated, `(LAMBDA () x)` returns `x`, which in this case is the value of y.

The existential formula created in Figure 26 is realized by the list of two elements: the first element is the realization of the function to compute the value of the existential variable y and the second element is the realization of the formula being quantified, in this case the disjunction.

$x = x$ by equintro();
{<<exis$R2:R-NONE>>}
$(x = x)$ or $(x \neq x)$ by disjintro(_PREVIOUS);
{<<exis$R3:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
B: ex y being Any st $(x = y)$ or $(x \neq y)$ by exintro(_PREVIOUS);
{<<exis.B$P:(R-LIST (LAMBDA () x) (LAMBDA () exis$R3)>>}

Figure 26: Existential Introduction on Formula with Content

In Mizar-C, the *consider* statement is used to eliminate an existential quantifier and introduce a new free identifier in the scope in which is it used. The type of the variable introduced by the *consider* statement must be the same as the type of the existential variable, and the formula given to hold for the new variable must match the original quantified formula. The use of the *consider* statement results in two realizations:

- the computation of the value of the variable being introduced.

  By *consider*ing the variable, we are requesting the value of the variable, which must now be computed due to the lazy semantics of the realization of the existential quantifier.

- the realization of the body of the existential formula.

Using existential statement *A* from Figure 25:

consider q being Any such that $(q = x)$ by direct(A);
{<<exis$R29:exis.A$P>>}
{<<exis.q$0:(R-APPLY0 exis$R29)>>}
{<<exis$R30:R-NONE>>}

Figure 27: Existential Elimination when Formula has no Content

The first realization computes the value of the existential variable by *unwrapping* the lambda expression. This is accomplished by the function `R-APPLY0`, which applies the lambda form to zero arguments. Since the body of the existential formula has no content, its realization is `R-NONE`.

Using statement *B* from Figure 26:

consider r being Any such that ((x = r) or (x ≠ r)) by direct(B);
{<<exis$R49:exis.B$P>>}
{<<exis.r$0:(R-APPLYO (R-FIRST exis$R49))>>}
{<<exis$R51:(R-APPLYO (R-SECOND exis$R49))>>}

Figure 28: Existential Elimination when Formula has Content

To compute the value of the variable, the funtion that computes it must be extracted from the list that realizes the existential statement. This is done using the function R-FIRST which returns the first element of a list. The function is then unwrapped as before. Since the body of the existential formula has content, its realization is the second element of the list, which is extracted using the R-SECOND function and then unwrapped.

B.1.9 Tuple Manipulation

Synopsis: *tuple[* Tuple *](* Formula *)*

The *tuple* inference rule provides a mechanism for accessing the components of a tuple. If the referenced tuple is a term, then a *Formula* where the term occurs positively must be given as justification. Given a *Tuple* ,the goal formula is an existential of the form

ex y1, y2, ..., yn st Tuple = <*y1, y2, ..., y*>

The reason for this crude manipulation is that the basic system has no built-in types that can be used for indexing into the data structure.

```
environ
  given t being <Any, Any, Any \verb¿+;
  {<<t:R-NONE>> }
  given Q being [<Any, Any > ];
  {<<R-NONE:R-NONE>> }
  given a being Any;
  {<<a:R-NONE>> }
  given f being (Any -> Any);
  {<<f:R-NONE>> }
  a1: Q[<a, (f a) > ];
  {<<a1$P0:R-NONE>>}
begin
  (ex q, r being Any st (<a, (f a) > = <q, r >) ) by tuple[<a, (f a) >](a1);
  {<<tuple$R5:(R-LIST
    (LAMBDA () (R-SELECT 0 (R-TUPLE a (FUNCALL f a))))
    (LAMBDA () (LAMBDA () (R-SELECT 1 (R-TUPLE a (FUNCALL f a)))) ) )>>}

    (ex x, y, z being Any st (<x, y, z > = t)) by tuple[t]();
    {<<tuple$R12:(R-LIST
      (LAMBDA () (R-SELECT 0 t))
      (LAMBDA () (R-LIST
        (LAMBDA () (R-SELECT 1 t))
         (LAMBDA () (LAMBDA () (R-SELECT 2 t)) ) )) )>>}
```

Figure 29: Example of Tuple Rule

B.1.10 Disjunction Manipulation

To introduce a disjunction the inference rule *disjintro* is used.
Synopsis: *disjintro* ($Formula_0, Formula_1, \ldots, Formula_m), m > 0$
Disjunction introduction allows one to weaken a statement by adding disjuncts to it.

$$\frac{\phi_0, \phi_1, \ldots, \phi_m}{(\theta_0 \vee \theta_1 \vee \ldots \vee \theta_n) \mid \theta}$$

where $1 < m \leq n$ and ϕ_i are $Formula_i$. All the formulae in the arguments must be represented in the goal disjunction, but the order of each ϕ_i in the goal is not important. The realization of a disjunction depends upon the realizations of each individual formula.

In disjunction A, the first disjunct is the one known to be true; in disjunction B, the second disjunct is the one known to be true. The disjunct whose value is unknown is represented by `R-NIL` in the realization.

```
1: x = x by equintro();
{<<disj.1$P:R-NONE>>}
A: (x = x) or (x ≠ x) by disjintro(1);
{<<disj.A$P:(R-LIST (LAMBDA () R-NONE) R-NIL)>>}
B: (x ≠ x) or (x = x) by disjintro(1);
{<<disj.B$P:(R-LIST R-NIL (LAMBDA () R-NONE))>>}
```

Figure 30: Disjunction Introduction

Disjunction Elimination is performed using the inference rule *disjelim*. A disjunct can be eliminated from a disjunction when we know the disjunct to be false.
Synopsis: *disjelim* (*Disjunction* [, $Formula_0, Formula_1, \ldots, Formula_m$]),$m \geq 0$
Disjunction elimination requires a disjunctive formula, *Disjunction*, and zero or more other argument formulae, $Formula_i$ which are the negations of the disjuncts to be eliminated. The goal may or may not be a disjunction.

$$\frac{\phi_0 \vee \phi_1 \vee \ldots \vee \phi_m, \theta_{P(0)}, \ldots, \theta_{P(n)}}{\gamma_{P(n+1)} \vee \gamma_{P(n+2)} \vee \ldots \vee \gamma_{P(m)} \mid \gamma_{P(m)}}$$

where $0 \leq n < m$ for some total 1-1 mapping $P : \{0, \ldots, m\} \rightarrow \{0, \ldots, m\}$ such that

$$\forall i \in \{0, \ldots, m\}, (\phi_i = \gamma_{P(i)}) \vee (\phi_i = \neg\theta_{P(i)}) \vee (\neg\phi_i = \theta_{P(i)}).$$

In other words, the goal is a disjunction like the first argument, except it is missing the disjuncts that are the negations of the formulae $\theta_{P(i)}$, and possibly has the disjuncts are rearranged. Using disjunctive formula *A* from Figure 30, disjunction elimination is performed.

```
now
   assume (x ≠ x);
   {<<disj$R1:disj$R1>>}
   thus contradiction by contrintro(1, _PREVIOUS);
   {<<disj$R2:R-NIL>>}
end;
{<<disj$R3:R-NONE>>}
C: (not (x ≠ x)) by negintro(_PREVIOUS);
{<<disj.C$P:R-NONE>>}
(x = x) by disjelim(A, C);
{<<disj$R6:R-NONE>>}
```

Figure 31: Disjunction Elimination

B.1.11 Conjunction Manipulation

In Mizar-C, a conjunction is considered to be a "bag" of formulae where the bag contents can be added to, subtracted from, and rearranged as long as all the formulae are true in the current context. One inference rule, *conj*, is used to manipulate conjunctive formulae.

Synopsis: *conj* $(Formula_0, Formula_1, \ldots, Formula_m), 0 \leq m$

$$\frac{\phi_0, \phi_1, \ldots, \phi_n}{(\theta_0 \& \theta_1 \& \ldots \& \theta_m) \mid \theta}$$

where θ_i is a well formed formula obtained from any of the $\phi_k, k = 1 \ldots n$, and θ is any well formed formula which is not a conjunction, obtained from any of the ϕ_i.

1: $(x = x)$ or $(x \neq x)$ by magic;
{<<conj.1$P:(R-LIST (LAMBDA () R-NONE) (LAMBDA () R-NONE))>>}
2: P[x];
{<<conj.2$P:R-NONE>>}
3: $((x = x)$ or $(x \neq x))$ & P[x] by conj(1,2):
{<<conj.3$P:conj.1$P>>}
4: P[x] & $((x = x)$ or $(x \neq x))$ by conj(1,2);
{<<conj.4$P:conj.1$P>>}
P[x] by conj(3);
{<<conj$R5:R-NONE>>}
$(x = x)$ or $(x \neq x)$ by conj(3);
{<<conj$R6:conj.3$P>>}

Figure 32: Examples of Conjunction Manipulation

The realization of each conjunction is dependent upon the realizations of its conjuncts, as described in Section A.1.

B.1.12 Induction

Induction is allowed on any type using any *well-founded partial order* for that type.
Synopsis: *induction(WellFoundedPredicate, InductionFormula)*

In Mizar-C, the form of the induction rule is that of strong induction, as follows:

$$\frac{\begin{array}{c}\mathit{WellFounded[TYPE,\ LT]},\\ \mathit{for\ x\ being\ TYPE\ holds\ (for\ y\ being\ TYPE\ holds\ LT[y,x]\ implies\ P[y])\ implies\ P[x]}\end{array}}{\mathit{for\ z\ being\ TYPE\ holds\ P[z]}}$$

In order to do induction over a type using a given ordering, a proof that the order is well-founded must be provided. The definitions and theorems described in Figure 5 are used to achieve this. For examples, see Section 4.4.

B.1.13 The Choice Rule

Synopsis: *choice(Formula)*
The choice rule allows the introduction of a function.

$$\frac{\mathit{for\ x\ being\ Tx\ holds\ (ex\ y\ being\ Ty\ st\ P[x,y])}}{\mathit{ex\ f\ being\ Tx \rightarrow Ty\ st\ (for\ x\ being\ Tx\ holds\ P[x,\ (f\ x)])}}$$

See Section 4.1 for further discussion. As implemented, the *choice* rule is inconvenient in that is does not return the fact the f is total. This becomes a problem when $(f\ x)$ does not occur positively in $P[\ ]$.

B.1.14 Guarded Choice

Synopsis: *gchoice(Formula)*
The guarded choice rule allows the introduction of a partial function.

$$\frac{\mathit{for\ x\ being\ Tx\ holds\ Guard[x]\ implies\ (ex\ y\ being\ Ty\ st\ P[x,y])}}{\mathit{ex\ f\ being\ Tx \rightarrow Ty\ st\ (for\ x\ being\ Tx\ holds\ Guard[x]\ implies\ P[x,\ (f\ x)])}}$$

See Section 4.2 for further discussion.

B.1.15 Definitions

The *define* construct in Mizar-C permits the definition of new predicates.

label : *define* Predicate Name *of* variable list *by* Formula ;

where variable list is of the following form:

variable names *being* variable type, ..., variable names *being* variable type

See Section 4.3 for further explanation.

B.2 Non-constructive Inference Rules

The rules in this section are inherently non-constructive, and never produce any formulae that have content.

B.2.1 Magic

Synopsis: *magic*

$$\frac{}{\theta}$$

The magic inference rule allows the system to accept any well-formed formula θ without references to another formula. The realization of the generated formula has the correct shape according to the realization rules for its type, but has no computation associated with it.

```
( P[x ] & P[y ] ) by magic();
{<<mag$R25:R-NONE>>}
( P[x ] or P[y ] ) by magic();
{<<mag$R26:(R-LIST (LAMBDA () R-NONE) (LAMBDA () R-NONE))>>}
(ex r being Any st (r = x)) by magic();
{<<mag$R29:(LAMBDA () $EX28) >>}
(ex r being Any st ( (r = x) or (r ≠ x) )) by magic();
{<<mag$R35:(R-LIST
  (LAMBDA () $EX34)
   (LAMBDA () (R-LIST (LAMBDA () R-NONE) (LAMBDA () R-NONE))))>>}
```

Figure 33: Use of Magic Inference Rule

B.2.2 Reverse Implication

In order to perform reverse implication elimination you must use the *revimpl* inference rule.

Synopsis: *revimpl(Implication | Iff , NegFormula)*

This rule is very similar to implication elimination, except it uses the negated conclusion of the implication to conclude the negated antecedent. It is also used for reverse iff elimination, where the *NegFormula* is the negation of either the left or right side of the iff formula.

$$\frac{\neg\theta, \phi \rightarrow \theta}{\neg\phi} \quad \frac{\neg\theta, \phi \leftrightarrow \theta}{\neg\phi} \quad \frac{\neg\phi, \phi \leftrightarrow \theta}{\neg\theta}$$

```
environ
  given P, Q being [Any ];
  {<<R-NONE:R-NONE>> <<R-NONE:R-NONE>> }
begin
  1: (P[x ] implies Q[x ]) by magic();
  {<<revimp.1$P:R-NONE>>}
  2: (not Q[x ]) by magic();
  {<<revimp.2$P:R-NONE>>}
  3: (not P[x ]) by revimpl(1, 2);
  {<<revimp.3$P:R-NONE>>}
  4: (P[x ] iff Q[x ]) by magic();
  {<<revimp.4$P:R-NONE>>}
  (not Q[x ]) by revimpl(4, 3);
  {<<revimp$R45:R-NONE>>}
  (not P[x ]) by revimpl(4, 2);
  {<<revimp$R46:R-NONE>>}
```

Figure 34: Examples of Reverse Implication Rule

B.2.3 Law of Excluded Middle

Synopsis: *exmiddle*
The Law of Excluded Middle allow us to create a new formula:

$$\overline{\theta \text{ or } \neg\theta | \neg\theta \text{ or } \theta}$$

where θ is any well-formed formula without any undefined variables.

```
( P[x ] or  (not P[x ]) ) by exmiddle();
{<<exmid$R52:R-NONE>>}
```

Figure 35: Excluded Middle Inference Rule

B.2.4 Negation Introduction

Synopsis: *negintro(* Implication*)*

$$\frac{\theta \rightarrow \texttt{contradiction}}{\neg\theta}$$

Negation introduction is used along with contradiction introduction for proving a negated formula θ, by assuming θ and proving that a contradiction follows.

```
now
  assume a: (x ≠ x);
   {<<negint$R60:negint$R60>>}
  (x = x) by eqintro();
  {<<negint$R57:R-NONE>>}
  thus contradiction by contrintro(a, _PREVIOUS);
  {<<negint$R65:R-NIL>>}
end; {<<negint$R68:R-NONE>>}
(not (x ≠ x)) by negintro(_PREVIOUS);
{<<negint$R69:R-NONE>>}
```

Figure 36: Example of Negation Introduction

B.2.5 Negation Elimination

Synopsis: *negelim(* Formula*)*

$$\frac{\neg\neg\theta}{\theta}$$

Negation Elimination reduces the number of *nots* in front of a formula by two.

B.2.6 Contradiction Introduction

Synopsis: *contrintro(* $Formula_0, \ldots, Formula_m$ *)* $m \geq 0$

$$\frac{\theta_0, \ldots, \theta_m}{contradiction}$$

Contradiction introduction is used in *now* reasonings in a "proof by contradiction" argument. The argument formulae must exhibit a contradictory pair of formulae; this means that either there are two formulae θ_i and θ_j that directly contradict each other, or else one of the θ_i is a conjunction where two of its conjuncts contradict each other. See Figure 36 for an example of use of this rule.

B.2.7 Contradiction

Synopsis: *contra(*contradiction*)*

$$\frac{contradiction}{\theta}$$

The *contra* rule is used to introduce any well-formed formula in the presence of *contradiction*.

```
now
   assume a: (x <> x);
   {<<cont$R94:cont$R94>>}
   (x = x) by eqintro();
   {<<cont$R98:R-NONE>>}
   contradiction by contrintro(_PREVIOUS, a);
   {<<cont$R101:R-NIL>>}
   thus P[x ] by contra(_PREVIOUS);
   {<<cont$R104:R-NONE>>}
end;
{<<cont$R107:R-NONE>>}
```

Figure 37: Example of Contra Rule

B.2.8 Equality Introduction

Synopsis: *eqintro(*Formula*)*
This rule allows the introduction of a term equality. If the equality is of a variable ϕ defined in the current context, then no reference *Formula* is necessary.

$$\frac{}{\phi = \phi}$$

The *equality* rule also allows the extraction of a term t from a *Formula* θ.

$$\frac{\theta}{t = t}$$

The goal formula is then an equality of a term t, and the referenced formula θ must make explicit positive mention of the term.

```
environ
  given P being [Any ];
  {<<R-NONE:R-NONE>> }
  given f being (Any -¿ Any);
  {<<f:R-NONE>> } given x being Any;
  {<<x:R-NONE>> }
  A: P[(f x) ];
  {<<A$P88:R-NONE>>}
begin
  (x = x) by eqintro();
  {<<eq$R89:R-NONE>>}

  ((f x) = (f x)) by eqintro(A);
  {<<eq$R90:R-NONE>>}
```

Figure 2: Equality Introduction

B.2.9 DeMorgan's Laws

Synopsis: *demorgan(*Formula*)*
The DeMorgan's Laws included in Mizar are very much like the classical ones.

$$\frac{\phi_0 \vee \phi_1 \vee \ldots \vee \phi_n}{\neg(\theta_{p(0)} \wedge \theta_{p(1)} \wedge \ldots \wedge \theta_{p(n)})} \quad \frac{\phi_0 \wedge \phi_1 \wedge \ldots \wedge \phi_n}{\neg(\theta_{p(0)} \vee \theta_{p(1)} \vee \ldots \vee \theta_{p(n)})}$$

$$\frac{\neg(\phi_0 \vee \phi_1 \vee \ldots \vee \phi_n)}{\theta_{p(0)} \wedge \theta_{p(1)} \wedge \ldots \wedge \theta_{p(n)}} \quad \frac{\neg(\phi_0 \wedge \phi_1 \wedge \ldots \wedge \phi_n)}{\theta_{p(0)} \vee \theta_{p(1)} \vee \ldots \vee \theta_{p(n)}}$$

where in each of the above, $\phi_i = \neq \theta_{p(i)}$ or $\neq \phi_i = \theta_{p(i)}$, for a permutation $p : \{0 \ldots n\} \to \{0 \ldots n\}$.

B.2.10 Conversion Between Disjunctions and Implications

There are two inference rules *imp2disj* and *disj2imp* that preserve the classical reasoning about the relationship between implication and disjunction.
Synopsis: *imp2disj(*Implication*)*
This rule transforms an implication into a disjunction.

$$\frac{\phi \to \theta}{(\neg\phi) \vee \theta \mid \theta \vee (\neg\phi)} \quad \frac{(\neg\phi) \to \theta}{\phi \vee \theta \mid \theta \vee \phi}$$

Synopsis: *disj2imp(*Disjunction*)*
This rule transforms a disjunction into an implication.

$$\frac{\phi \vee \theta}{(\neg\phi) \to \theta} \quad \frac{(\neg\phi) \vee \theta}{\phi \to \theta}$$

```
environ
  given P, Q being [ ];
  {<<R-NONE:R-NONE>> <<R-NONE:R-NONE>> }
begin
  1: (not ( P[ ] or Q[ ] )) by magic();
  {<<demorg.1$P:R-NONE>>}
  ( (not P[ ]) & (not Q[ ]) ) by demorgan(1);
  {<<demorg$R30:R-NONE>>}
  2: (not ( P[ ] or (not Q[ ]) )) by magic();
  {<<demorg.2$P:R-NONE>>}
  ( (not P[ ]) & Q[ ] ) by demorgan(2);
  {<<demorg$R31:R-NONE>>}
  3: ( (not P[ ]) or (not Q[ ]) ) by magic();
  {<<demorg.3$P:(R-LIST (LAMBDA () R-NONE) (LAMBDA () R-NONE))>>}
  (not ( P[ ] & Q[ ] )) by demorgan(3);
  {<<demorg$R33:R-NONE>>}
```

Figure 39: Examples of Demorgan Rule

```
( (not P[ ]) or P[ ] ) by magic();
{<<impdisj$R39:(R-LIST (LAMBDA () R-NONE) (LAMBDA () R-NONE))>>}
(P[ ] implies P[ ]) by disj2imp(_PREVIOUS);
{<<impdisj$R41:R-NIL>>}
( (not P[ ]) or P[ ] ) by imp2disj(_PREVIOUS);
{<<impdisj$R42:R-NIL>>}
```

Figure 40: Conversion between Implication and Disjunction

Appendix C

Basic Bit String Extensions

The following information about the basic bit string data type has been added to Mizar-C:

given Bits being [Any];
{<<R-NONE:R-NONE>> }
given 0, 1, nil being Bits rby bit-0, bit-1, bit-nil;
{<<bit-0:bit-0>> <<bit-1:bit-1>> <<bit-nil:bit-nil>> }
given cat being (<Bits, Bits > -> Bits) rby bit-cat;
{<<bit-cat:bit-cat>> }
given split being (<Bits, Bits > -> <Bits, Bits >) rby bit-split;
{<<bit-split:bit-split>> }
given bits_len_lt being [Bits, Bits];
{<<R-NONE:R-NONE>> }
given bits_len_eq being [Bits, Bits];
{<<R-NONE:R-NONE>> }
given LT being [Bits, Bits];
{<<R-NONE:R-NONE>> }
BA_nil_not_1: (nil <> 1) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_nil_not_0: (nil <> 0) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_1_not_0: (1 <> 0) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_nil_or_not: (for x being Bits holds ((x = nil) or (x <> nil))
) rby bits-ba-nil-or-not;
{<<bits-ba-nil-or-not:bits-ba-nil-or-not>>}
BA_len_lt_nil: (for x being Bits holds
((x <> nil) iff bits_len_lt[nil, x])) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_len_lt_asym: (for x, y being Bits holds
(bits_len_lt[x, y] implies (not bits_len_lt[y, x]))) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_len_lt_trans: (for x, y, z being Bits holds (bits_len_lt[x, y] implies

(bits_len_lt[y, z] implies bits_len_lt[x, z]))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_len_not_lt: (for x, y being Bits holds ((not bits_len_lt[x, y]) implies
(bits_len_lt[y, x] or bits_len_eq[x, y]))) *rby bits-ba-len-not-lt;*
{<<bits-ba-len-not-lt:bits-ba-len-not-lt>>}
BA_len_eq_no_lt: (for x, y being Bits holds (bits_len_eq[x, y] iff
(not (bits_len_lt[x, y] or bits_len_lt[y, x])))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_len_eq_1: (for x being Bits holds
(bits_len_eq[1, x] iff ((x = 1) or (x = 0)))) *rby bits-ba-len-eq-1;*
{<<bits-ba-len-eq-1:bits-ba-len-eq-1>>}
BA_len_eq_refl: (for x being Bits holds bits_len_eq[x, x]) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_len_eq_sym: (for x, y being Bits holds
(bits_len_eq[x, y] implies bits_len_eq[y, x])) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_len_eq_trans: (for x, y, z being Bits holds (bits_len_eq[x, y] implies
(bits_len_eq[y, z] implies bits_len_eq[x, z]))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat_nil: (for x being Bits holds
(((cat <nil, x >) = x) & ((cat <x, nil >) = x))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat_len_eq: (for x, y, z being Bits holds (bits_len_eq[y, z] iff
bits_len_eq[(cat <x, y >), (cat <x, z >)])) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat_len_lt: (for x, y, z being Bits holds (bits_len_lt[y, z] iff
bits_len_lt[(cat <x, y >), (cat <x, z >)])) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat_both_len_eq: (for x, y being Bits holds
bits_len_eq[(cat <x, y >), (cat <y, x >)]) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat: (for x, y being Bits holds
(ex z being Bits st (z = (cat <x, y >)))) *rby bits-ba-cat;*
{<<bits-ba-cat:bits-ba-cat>>}
BA_cat_assoc: (for x, y, z being Bits holds
((cat <(cat <x, y >), z >) = (cat <x, (cat <y, z >) >))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_cat_split: (for x, y being Bits holds
(x = (cat (split <x, y >)))) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_split_cat: (for x, y being Bits holds
((split <(cat <x, y >), x >) = <x, y >)) *rby R-NONE;*
{<<R-NONE:R-NONE>>}
BA_split: (for x, y being Bits holds
(ex z1, z2 being Bits st ((split <x, y >) = <z1, z2 >))) *rby bits-ba-split;*
{<<bits-ba-split:bits-ba-split>>}

BA_split_1: (for x being Bits holds (((split <1, x >) = <1, nil >) or
((split <1, x >) = <nil, 1 >))) rby bits-ba-split-1;
{<<bits-ba-split-1:bits-ba-split-1>>}
BA_split_eq: (for x, y, z being Bits holds (bits_len_eq[y, z] implies
((split <x, y >) = (split <x, z >)))) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_split_eq_rev: (for x, y, z being Bits holds
((not bits_len_lt[x, y]) implies ((not bits_len_lt[x, z]) implies
(((split <x, y >) = (split <x, z >)) iff
bits_len_eq[y, z]))))) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_split_big: (for x, y being Bits holds
((not bits_len_lt[y, x]) iff ((split <x, y >) = <x, nil >))) rby R-NONE;
{<<R-NONE:R-NONE>>}
BA_LT: (for x, y being Bits holds
(bits_len_lt[x, y] iff LT[x, y])) rby R-NONE;
{<<R-NONE:R-NONE>>}

Bibliography

[1] S. Berardi. Pruning simply typed lambda-terms. Technical report, Dipartimento di Informatica dell'Universita' di Torino (University of Turin, Italy).

[2] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5:127–139, May 1989.

[3] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] Thierry Coquand. On the analogy between propositions and types. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 399–418, Reading, Massachusetts, 1990. Addison-Wesley.

[5] J. N. Crossley and J. B. Remmel. Proofs, programs and run-times, October 1991.

[6] G. Dowek et al. *The* COQ *Proof Assistant User's Guide*, February 1992.

[7] P. Dybjer. Program verification in a logical theory of constructions. *Lecture Notes in Computer Science*, 201:334–349, 1985.

[8] J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[9] J. Gallier. On the correspondence between proofs and lambda terms, January 1993.

[10] Susumu Hayashi and Hiroshi Nakano. PX*: A Computational Logic*. The MIT Press, Cambridge, Masschussets, 1989.

[11] E. C. R. Hehner. *a Practical Theory of Programming*. University of Toronto, Department of Computer Science, 1992. Draft.

[12] M. C. Henson. Realizability models for program construction. *Lecture Notes in Computer Science*, 375:256–272, June 1989. Proc. Math of Prog. Constr.

[13] M. C. Henson and R. Turner. A constructive set theory for program development. *Lecture Notes in Computer Science*, 338:329–347, 1988. Proc. 8th conf. on FST & TCS.

[14] H. J. Hoover and P. Rudnicki. *Introduction to Logic in Computing Science.* University of Alberta, Department of Computing Science, 1993.

[15] S.C. Kleene. *Introduction to Metatmathematics.* Van Nostrand, 1952.

[16] B. Knight. Safe strict evaluation of redundancy-free programs from proofs. Master's thesis, University of Victoria, 1994.

[17] P. A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, pages 3–27, January 1988.

[18] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Sofware Engineering*, 18(8):674–704. August 1992.

[19] P. Martin-Lof. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice/Hall, 1985.

[20] J.S. Moore et al. Special issue on system verification. *Journal of Automated Reasoning*, 5:409–530, April 1989.

[21] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, August 1990.

[22] N. N. Nepejvoda. A bridge between constructive logic and computer programming. *Theoretical Computer Science*, 90(1):253–270, November 1991.

[23] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94:335–356, 1992.

[24] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin-Heidelberg-New York, 1989.

[25] P. Rudnicki, Y. Nakamura, and A. Trybulec. Articles AMI_1 .. AMI_5 SCM_1, Mizar Data Base, follow directions in http://web.cs.ualberta.ca/~piotr.

[26] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1989.

[27] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction, vols I and II*, volume 121 & 123 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1988.

[28] A. Voronkov. Higher order functions in first order logics. *Lecture Notes in Computer Science*, 601, May 1992.

[29] A. Walenstein, J. H. Hoover, and P. Rudnicki. Programming with constructive proof in the MIZAR-C proof environment. Technical Report 92-12, University of Alberta, 1992.